Programación con Visual Basic .NET

Francisco Charte

INCLUYE CD-ROM con multitud de ejemplos listos para utilizar

Aprenda a usar formularios Windows y Web usando los diseñadores de Visual Basic .NET

Adquiera las bases necesarias para utilizar GDI+, ADO .NET, ASP .NET y los demás servicios de la plataforma .NET

Conozca el nuevo lenguaje Visual Basic .NET, y sus características de orientación a objetos y componentes

ANAYA MULTIMEDIA

Programación con

Visual Basic .NET

Francisco Charte Ojeda

ANAYA MULTIMEDIA

Vuestra sonrisa es mi alegría diaria,
mi energía, las ganas de vivir.
Me basta con miraros a los ojos para sentir
que el Sol brille más y el cielo se torne más azul.
Gracias por darme tanto cariño y tanto amor
y conseguir que el mundo parezca mejor.

A mis hijos David y Alejandro, o viceversa.

Agradecimientos

Por costumbre, y también algo de lógica, estas páginas, que están al principio del libro y que no tienen el más mínimo interés para la mayoría de los lectores, se redactan siempre al final, prácticamente cuando todo el trabajo, no sólo el del autor, ha llegado a su fin. Su redacción es como una marca divisoria: se termina, por fin, el duro trabajo de muchas semanas y personas y se avista, por suerte, el producto que finalmente llegará a manos del lector.

El autor puede tomarse un respiro y dejar descansar el teclado del ordenador, y dedicar unos minutos a recorrer todo el proceso que ha seguido hasta llegar aquí. Primero la idea de elaborar este libro, surgida de mis editores, Víctor Manuel y Susana, y el que suscribe. Les agradezco, y por extensión a toda la editorial, la confianza que depositan reiteradamente en mí. De verdad, espero seguir mereciéndola y no fallaros.

Para que las ideas puedan ser convertidas en algo tangible hacen falta unas herramientas, sin las cuales sería imposible desarrollar nuestro trabajo. Esas herramientas no las habríamos tenido en nuestras manos sin la inestimable colaboración y ayuda de Microsoft Ibérica, personificada en mi caso en las personas de Javier Izquierdo, José Antonio Álvarez e Ibrahim Pérez. Gracias por esa ayuda y por poner a nuestro alcance estos impresionantes productos.

¿Qué sería de la vida sin los amigos? Éstos habitualmente no suelen intervenir en nuestro trabajo pero, por suerte, hay excepciones: los amigos-compañeros-consejeros. Por una parte quiero agradecer el apoyo y la ayuda, siempre dirigida a mejorar la calidad del libro, de Tomás Eisman que, además, es la persona que se ocupa de convertir un montón de documentos y archivos de imágenes en un libro de verdad, tal y como llega a las manos del lector. También doy las gracias a Jorge Serrano, un amigo, primero, y compañero en Anaya Multimedia, después, siempre dispuesto a ayudar en lo que esté en su mano. Gracias por tu apoyo y gastar tanto en teléfono para charlar conmigo.

Por último, y, como suele decirse, no por ello menos importante, el agradecimiento más profundo a mi familia, las personas que me animan minuto a minuto y que sufren esta afición-dedicación que no entiende de horas ni fines de semana. Os quiero.

Índice de contenidos

Agradecimientos 6

Introducción 23

Posibilidades de Visual Basic .NET 24
Objetivo de este libro 25

1. Primera toma de contacto 29

Puesta en marcha de Visual Basic .NET 29
La página de inicio 30
Elementos fundamentales 32
Inicio de un nuevo proyecto 33
Análisis del código 34
Ejecución 36
Un vistazo al ensamblado 37
Opciones del proyecto 38
Puntos clave 39
Resumen 40

2. El entorno de desarrollo 43

Disposición general 43
Soluciones y proyectos 45

Objetos existentes en un módulo 47
Examinando clases y otros elementos 48
Acceso a los asistentes 49
Diseñadores y editores 50
Diseño de formularios 51
Edición de propiedades 52
Componentes y controles 54
Diseñadores adicionales 56
El editor de código 58
Otras herramientas 59
El Explorador de servidores 59
Tareas pendientes 60
La ventana de comandos 61
La ayuda de Visual Basic .NET 62
Adaptación del entorno 62
Opciones del entorno 64
Configuración de botones y menús 65
Selección de paletas visibles 65
Añadir y eliminar botones 65
Añadir y eliminar menús de opciones 67
Creación de nuevas paletas y menús 69
Opciones diversas 69
Configuración de herramientas externas 70
Automatización de tareas 72
Registro de una nueva macro 73
Ejecución de una macro 73
Código de la macro 74
Puntos clave 74
Resumen 76

3. La plataforma Microsoft .NET 79

Nuevas necesidades 80
.NET, ¿la solución definitiva? 81
Esquema de desarrollo y ejecución 82
Composición de una aplicación .NET 83
Ensamblados y manifiestos 84
Ensamblados y dominios de aplicación 86
Ensamblados privados y compartidos 87
Proceso de compilación y ejecución 88
Código intermedio 89
El sistema común de tipos 89
Ejecución supervisada 91
Compilación JIT 92

Información de tipos 92
Servicios .NET para las aplicaciones 93
Puntos clave 94
Resumen 95

4. Tipos de datos 97

El sistema común de tipos 97
Un sistema de tipos unificado 98
Empaquetado y desempaquetado de datos 99
Tipos intrínsecos 101
Tipos no ajustados a la CLS 103
Declaración de variables 105
Ámbitos y visibilidad 105
Vida de una variable 107
Especificación de tipo 109
Asignación de un valor inicial 109
Tipos definidos por el usuario 110
Enumeraciones 110
Comprobación estricta de tipos 111
Estructuras 113
Asignación e igualdad 114
Miembros públicos y privados 115
Métodos, constructores y propiedades 116
Arreglos 117
Declaración de un arreglo 118
Arreglos con varias dimensiones 119
Rangos dinámicos 119
Información sobre un arreglo 121
Asignación y copia 122
Otras operaciones con arreglos 124
Trabajo con caracteres 125
Cadenas de caracteres 126
Manipulación intensiva de cadenas 126
Constantes 128
Puntos clave 129
Resumen 130

5. Operadores y expresiones 133

Asignación de valores 133
Operadores aritméticos 134
Operaciones con variables Object 135
Operadores relacionales 136

Operadores lógicos 137
Operadores de circuito corto 138
Operaciones entre bits 140
Expresiones con referencias 142
Puntos clave 143
Resumen 144

6. Estructuras de control 147

Estructuras condicionales 147
Varias sentencias en una línea 148
La salida falsa 149
Condiciones anidadas 149
Condiciones que siempre devuelven un valor 150
Condicionales múltiples 152
Estructuras de repetición 154
Bucles por contador 154
Bucles condicionales 155
Bucles que recorren un arreglo 156
Control estructurado de excepciones 157
Procedimientos y funciones 160
Declaración de un procedimiento 161
Recepción de parámetros 161
Devolución de valores 162
Devolución de arreglos 163
Parámetros por valor y por referencia 164
Parámetros opcionales 165
Salida de un procedimiento 167
Procedimientos recursivos 168
Llamadas a métodos por nombre 169
Parámetros al usar CallByName() 169
Puntos clave 170
Resumen 171

7. Programación orientada a objetos 173

Ámbitos con nombre 173
Definición de ámbitos con nombre 174
Referencias a un ámbito con nombre 176
Conflictos entre ámbitos con nombre 177
Ámbitos con nombre en Visual Basic .NET 181
Clases de objetos 181
Definición de una clase 182
Visibilidad de una clase y sus miembros 182

Clases derivadas y herencia 183
Constructores y destructores 186
Clases anidadas 188
Objetos de una clase 189
Miembros de una clase 190
Miembros compartidos 190
Miembros sobrecargados 193
Miembros redefinidos y ocultos 194
Acceso a los miembros de la clase base 195
Uso polimórfico de los objetos 197
Interfaces 200
Definición de una interfaz 201
Implementación de una interfaz 201
Módulos 203
Puntos clave 203
Resumen 204

8. Programación orientada a componentes 207

Propiedades 208
Definición de una propiedad 208
Propiedades de sólo lectura y sólo escritura 209
Propiedades con índices 211
La propiedad por defecto 212
Delegados y eventos 213
Definición implícita del delegado 215
Asociación entre eventos y controladores 216
Otros usos de los delegados 217
Componentes .NET 219
Clases y componentes 220
Uso no visual 222
Uso visual del componente 224
Puntos clave 227
Resumen 227

9. Programación concurrente 229

Aplicaciones de la concurrencia 229
Aplicaciones, hilos y prioridades 231
La clase Thread 231
Puesta en marcha del nuevo hilo 232
Enumeraciones concurrentes 233
Prioridad de ejecución 235
Elementos de sincronización 236

Esperas entre hilos 238
Accesos exclusivos 239
Otros métodos de sincronización 241
Interbloqueos y señales 242
Puntos clave 243
Resumen 243

10. Formularios Windows 247

¿Qué son los formularios Windows? 248
Una aplicación Windows sencilla 248
Diseño del formulario 249
Implementación de funcionalidad 249
Análisis del código generado 250
Resumiendo 256
Información sobre la aplicación 257
Inicio de la aplicación 257
Datos del entorno de la aplicación 258
Otros miembros de interés 259
Aplicaciones de documento múltiple 260
Preparación de la ventana principal 261
Creación de un menú de opciones 261
Diseño de la ventana hija MDI 262
Unión de las piezas 264
Trabajo con componentes 265
Miembros comunes 265
Posición y dimensiones 266
Orden de acceso y foco de entrada 267
Anclaje y adosado de controles 268
Eventos de teclado y ratón 271
Arreglos de controles 275
La colección de controles 276
Uso compartido de gestores de eventos 277
Creación de controles en ejecución 278
Herencia visual 279
Puntos clave 280
Resumen 281

11. Formularios Web 285

De ASP a ASP.NET 286
Naturaleza de un documento ASP 286
Ocultación del código en ASP.NET 287
Libertad de elección de lenguaje 289

Adiós a los guiones interpretados 289
Enlace entre el documento y el código 292
El diseñador de formularios Web 293
La clase Page 293
Distribución del contenido 295
Eventos y gestores 295
Componentes HTML 296
Componentes de servidor 296
Uso de componentes de servidor 297
Examen del código HTML en el servidor 298
Examen del código en el cliente 300
Componentes de validación 300
Solicitudes y respuestas 302
Identificación del cliente 303
Parámetros de la solicitud 304
Parámetros de formularios 306
Uso de cookies 309
Módulos de configuración 310
Edición del archivo web.config 311
Almacenamiento de datos de aplicación 311
Recuperando información de configuración 312
Puntos clave 313
Resumen 315

12. Servicios de entrada y salida 317

Primera aproximación 318
El paradigma de los flujos de datos 318
Preparados para otras plataformas 319
Recuperando información del sistema de archivos 319
Unidades existentes en el sistema 319
Enumeración de carpetas y archivos 320
Información sobre un objeto 321
Unidades, caminos y nombres 322
Manipulación de carpetas y archivos 325
Acceso al contenido de los archivos 325
Metodología general 326
Apertura de archivos 327
Datos con tipo 330
Un visor de archivos de texto 331
Flujos binarios 335
Flujos en memoria 336
Puntos clave 337
Resumen 338

13. Elaboración de gráficos 341

Ámbitos relacionados con GDI+ 342
Elementos de propósito general 343
Gráficos vectoriales y 2D 343
Manipulación de imágenes 344
Impresión 344
Tipografías 345
Primera aproximación 345
Preparación de un pincel 346
Dibujo de entidades simples 346
Relleno de áreas 347
Añadimos un texto 348
Elementos básicos 349
Colores 350
Puntos y coordenadas 353
Brochas 355
Lápices 357
Tipos de letra 362
Manipulación de mapas de bits 363
Recuperación de una imagen 364
Información y manipulación 364
Visualización de las imágenes 365
Caminos y regiones 368
Creación de un camino 369
Transformaciones 370
Creación de una región 371
El componente PictureBox 372
Gráficos en ASP.NET 373
Puntos clave 374
Resumen 376

14. Impresión de datos 379

Fases del proceso de impresión 380
Documento a imprimir 380
Superficie de impresión 381
Composición de un documento 382
Configuración de parámetros 387
Selección del dispositivo de destino 388
Configuración de página 390
Visualización preliminar 391
Puntos clave 392
Resumen 393

15. Desarrollo de componentes 397

Conceptos básicos 398
¿Cuándo un objeto es un componente? 398
Un sencillo ejemplo 399
Instalación del componente 402
Uso visual del componente 404
Mejoras para la fase de diseño 405
Atributos para componentes 405
Actualización automática de la ventana Propiedades 407
Valores por defecto y persistencia 408
Validez de los valores asignados 409
Eventos y atributos 410
Editores y convertidores a medida 413
Puntos clave 414
Resumen 414

16. Controles Windows 417

Tipos de controles Windows 417
La clase Control 419
Derivar de Control 419
Dibujo del control 420
Propiedades protegidas 422
Ocultación de propiedades 425
Eventos de cambios en propiedades 426
Nuevas propiedades y eventos 428
Interceptar los eventos de teclado y ratón 430
Controles compuestos 430
Propiedades de componentes contenidos 433
Otras bases para nuestros controles 435
Puntos clave 437
Resumen 438

17. Controles Web 441

La clase Control y sus derivadas 441
Desarrollo de un control simple 442
Posición, dimensiones y estilo 443
Adición de nuevos miembros 444
La clase HtmlTextWriter 446
Controles compuestos 448
Pagelets 451
Creación de un pagelet 451

Uso de un pagelet 452
Puntos clave 454
Resumen 455

18. Creación de servicios Windows 457

Esquema general 458
La clase ServiceBase 460
Creación del objeto que actúa como servicio 460
Eventos operacionales 461
Instalación del servicio 461
Asistentes disponibles para desarrollar servicios 462
Un servicio horario para red 464
Definición de la clase de servicio 465
Preparación del instalador 467
Instalación del servicio 468
Comprobación del servicio 469
Controlando los servicios 469
La clase ServiceController 470
Un controlador genérico 471
Un controlador para Serviciol lora 476
Puntos clave 480
Resumen 481

19. Desarrollo de servicios Web 483

¿Qué es un servicio Web? 483
Servicios y aplicaciones 484
Universalidad de un servicio 484
Protocolos y lenguajes 485
Introducción a XML 487
¿Qué es XML? 487
Trabajo con etiquetas 488
Visualización de los documentos 489
Estructura de un documento XML 492
Estructura de los datos 492
Conjuntos de caracteres en XML 493
Documentos XML bien formados 495
Documentos XML válidos 496
Analizadores de XML 497
Validación de documentos 497
¿Qué es una DTD? 499
Análisis de la estructura del documento 499
Elaboración de la DTD 500

Cómo aplicar una DTD a un documento XML 502
Validación del documento 503
Atributos o propiedades 506
Introducción a XSL 507
Transformación de documentos 507
XSL básico 508
Cómo aplicar plantillas 509
Selección de datos 510
Las posibilidades de XSL 514
Introducción a XSD 515
El editor de esquemas XML de Visual Basic .NET 516
Codificación manual del esquema 517
Introducción a XSLT y XPath 518
Introducción a SOAP 520
Estructura de un mensaje SOAP 520
SOAP y Visual Basic .NET 521
Introducción a WSDL y UDDI 521
Creación de un servicio Web 522
Anatomía de un servicio Web 523
El servicio horario 523
El módulo asmx 524
Modulo de descripción del servicio 525
Consumo de un servicio Web 530
Referencias externas 532
¿Dónde esta el proxy? 534
Uso del servicio Web desde el consumidor 535
Puntos clave 536
Resumen 537

20. Fundamentos de tratamiento de datos 539

¿Qué es una base de datos? 540
¿Qué es un RDBMS? 540
Estructura física de una base de datos 541
Nombres de archivo 541
Estructura lógica de una base de datos 542
Tablas 543
Índices 544
Vistas 544
Procedimientos almacenados 544
Otros elementos lógicos 545
Introducción a SQL 546
El Analizador de consultas SQL 546
Componentes y derivados de SQL 547

DDL 548
DML 548
Transact-SQL 548
Definición de datos 549
Creación de una base de datos 549
Creación de tablas 551
Modificación de una tabla 552
Otras operaciones 554
Manipulación de datos 554
Inserción de nuevas filas 555
Selección de datos 556
Modificación de datos 557
Eliminación de datos 558
Relaciones entre tablas 559
Puntos clave 560
Resumen 561

21. Acceso a datos con ADO.NET 565

Modelo de objetos de ADO.NET 566
Estructura de un objeto DataSet 566
Creación de un DataSet a partir de una tabla 568
Recuperación de la estructura de un DataSet 568
Control de la conexión 570
Acceso a los datos 571
Actualización de los datos 572
Enlace de datos e interfaz 574
Enlace en aplicaciones Web 575
Enlace en aplicaciones Windows 577
La colección DataBindings 578
Asistentes y diseñadores de Visual Basic .NET 579
Creación automática de objetos 580
El asistente para formularios de datos 583
Puntos clave 586
Resumen 587

22. Diseño de informes con Crystal Reports .NET 589

El diseñador de informes 590
Creación del informe 590
Modificación del diseño 592
Fórmulas y campos especiales 594
Formato de los elementos 596
Explotación del informe 597

Visualización en un formulario Windows 597
Visualización en un formulario Web 598
Impresión y exportación del informe 600
Puntos clave 604
Resumen 604

23. Recuperación de información de tipos en ejecución 607

Servicios de reflexión 608
Ensamblados y módulos 608
Información sobre el ensamblado 609
Módulos que componen el ensamblado 610
Tipos definidos en un ensamblado 612
La clase Type 612
Parámetros de un método 613
Jerarquía de tipos de un ensamblado 614
Uso dinámico de objetos 617
La clase Activator 617
Invocación dinámica 618
Puntos clave 620
Resumen 620

24. Interactuación con COM y el API de Windows 623

Uso de componentes COM 624
Importación de controles ActiveX 624
Importación de librerías de tipos 625
Uso dinámico de componentes COM 627
Exposición de componentes .NET a COM 630
Desarrollo y opciones del componente 631
Uso del componente desde un cliente COM 632
Acceso al API de Windows 634
Declaración de la función 634
Uso de las funciones 635
Puntos clave 635
Resumen 635

25. Conversión de proyectos Visual Basic 6 a Visual Basic .NET 639

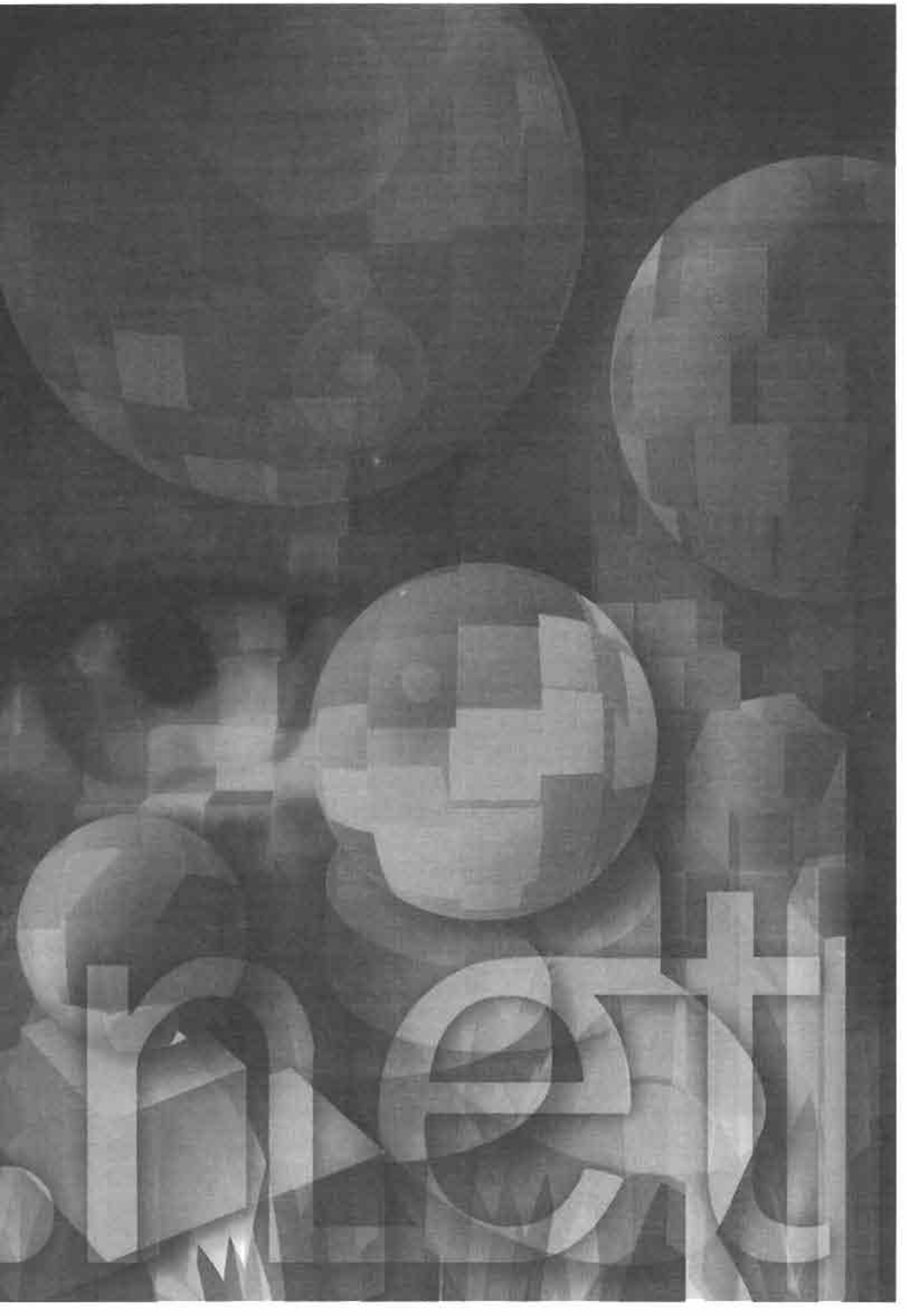
El asistente para actualización 640
Inicio del proceso 641
Informe de actualización 642
La capa de compatibilidad de Visual Basic 642

Controles y llamadas al API 643
Cambios obligados 644
Puntos clave 645
Resumen 645

A. Contenido del CD-ROM 649

Uso de los ejemplos 649
Atención al lector 650

Índice alfabético 651



Introducción

Hace una década, a principios de los noventa, Microsoft revolucionó el desarrollo de aplicaciones para Windows con la presentación de la primera versión de Visual Basic, un lenguaje que, con el tiempo, se ha convertido en el más utilizado por los programadores que tienen el sistema operativo Windows como objetivo. Hasta entonces, las opciones para construir aplicaciones para este sistema se limitaban prácticamente al uso del lenguaje C y la compleja API de Windows.

Visual Basic simplificó de manera sorprendente tareas que, hasta ese momento, resultaban arduas. El simple diseño de un cuadro de diálogo, conteniendo listas, cajas de texto y botones, requería un trabajo considerable utilizando C. Con Visual Basic, por el contrario, era una tarea acometida con operaciones de arrastrar y soltar y que podía resolverse en minutos.

En esta década, desde Visual Basic 1.0 hasta la presentación de Visual Basic .NET, no es necesario decir que todos los aspectos que tienen que ver con la informática han cambiado mucho y, obviamente, el desarrollo de aplicaciones no es un campo ajeno a esos cambios. Ahora los proyectos no son desarrollados de forma monolítica por un mismo equipo, sino que se usan componentes discretos, propios u ofertados por terceros, para crear aplicaciones distribuidas que, en muchas ocasiones, usan Internet como plataforma de trabajo.

Las herramientas y lenguajes de entonces no están, en su mayor parte, preparados para estas nuevas necesidades, lo cual implica un trabajo mucho mayor por parte de los programadores. Suponga, por ejemplo, que quiere crear un componente transaccional y con capacidad de atender a múltiples clientes, con varios hilos de ejecución paralelos, elaborando la información de una base

de datos para facilitarla a unas interfaces Windows y Web. Actualmente, para crear el componente sería preciso utilizar Visual C++, para la interfaz Windows lo más fácil sería recurrir a Visual Basic y para la interfaz Web a Visual InterDev. Si necesitásemos otras posibilidades, como la de ofrecer la misma función en forma de servicio Web, deberíamos recurrir a añadidos a Visual Studio 6.

Posibilidades de Visual Basic .NET

La versión definitiva de Visual Basic .NET será presentada en San Francisco el próximo día 13 de febrero de 2002, como parte de Visual Studio .NET. En el momento de escribir este libro, a finales de 2001, está disponible tanto una versión *beta* en castellano de Visual Studio .NET como una *Release Candidate* en inglés. El producto Visual Basic .NET, sin embargo, no existe actualmente como tal, separadamente, sino como parte de Visual Studio .NET, aunque sí lo estará posteriormente ya de manera comercial. Para escribir este libro se ha utilizado la versión *beta* 2 en castellano de Visual Studio .NET *Enterprise Architect*, de la que forman parte también Visual C# y Visual C++ .NET.

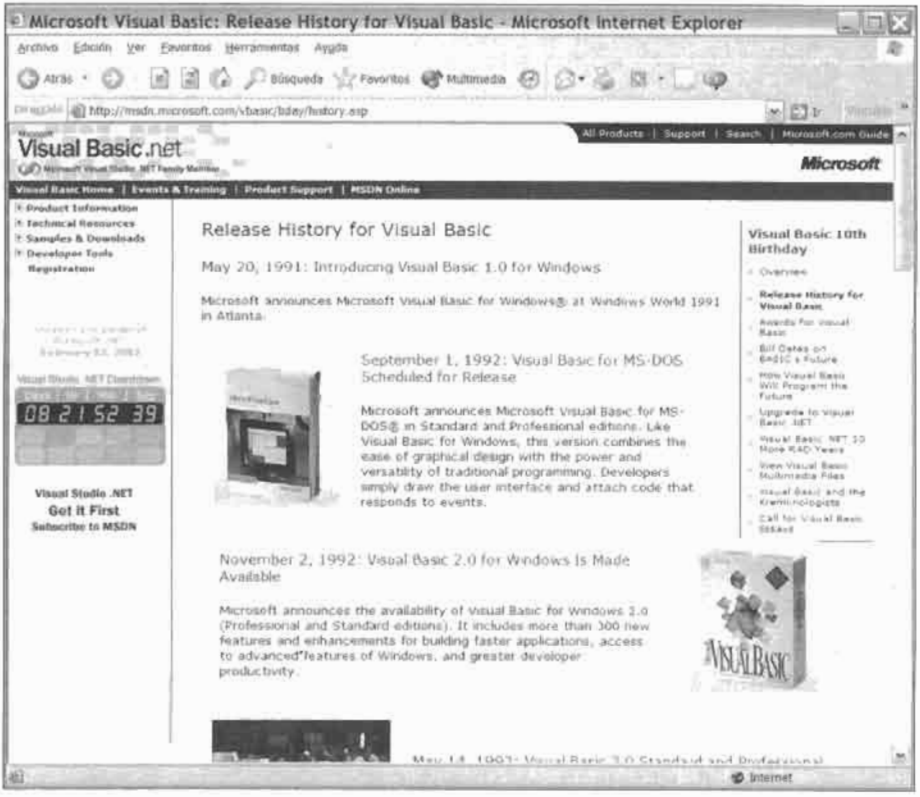


Figura 1.1. En la sede de Microsoft podemos encontrar la historia y evolución de Visual Basic

Independientemente de esta circunstancia, y de que adquiramos Visual Studio .NET completo o Visual Basic .NET de manera individual, las cualidades que encontraremos serán prácticamente las mismas. Visual Basic es ahora un lenguaje orientado a objetos, con cualidades similares a las de C++, sin por ello haber perdido su especialidad: la manipulación y creación de componentes. Podemos crear clases en Visual Basic .NET derivándolas de otras, ya estén escritas también en Visual Basic .NET o bien en otros lenguajes .NET, como C#. Disponemos, por tanto, de una característica largo tiempo reclamada por una parte de los usuarios de Visual Basic: la herencia.

También es un lenguaje que cuenta con la capacidad de *sobrecargar* métodos, de tal forma que pueden existir múltiples versiones de un mismo método, cada uno tomando una lista de parámetros distinta. Se ha añadido un control estructurado de excepciones moderno, dando paso a la superación del arcaico `On Error Goto`. Existen operadores aritméticos y lógicos abreviados, inicialización de variables durante la declaración, mayor consistencia de tipos de datos, etc.

Cualquier tipo de aplicación escrita con Visual Basic .NET puede utilizar libremente varios hilos de proceso concurrente, lo que habitualmente se denomina *free threading*. No sólo podemos crear aplicaciones para Windows y componentes para ellas, también podemos usar Visual Basic .NET para producir aplicaciones de consola, servicios Windows, interfaces Web, componentes de servidor ASP.NET y servicios Web.

Como lenguaje integrado completamente en la plataforma .NET, Visual Basic .NET genera código MSIL (*Microsoft Intermediate Language*) y, por tanto, tiene un acceso total a los servicios de dicha plataforma. Esto significa que puede utilizar centenas de clases de objetos y componentes para, por ejemplo, manipular archivos en disco, obtener información de tipos en ejecución, trabajar con gráficos, acceder a bases de datos, etc. Todos esos servicios se engloban en algunos grupos, como ADO.NET, para el acceso a datos; GDI+, para los gráficos e impresión; ASP.NET, para todo lo relacionado con la Web, etc.

En definitiva, con Visual Basic .NET no necesitaría recurrir a ningún otro lenguaje ni herramienta externa para construir la aplicación distribuida que se indicaba al final del punto previo. Visual Basic .NET es un lenguaje mucho más potente y flexible, un lenguaje orientado a objetos y al trabajo con componentes que, prácticamente, se pone al mismo nivel que C++ pero sin las complejidades propias de dicho lenguaje.

Objetivo de este libro

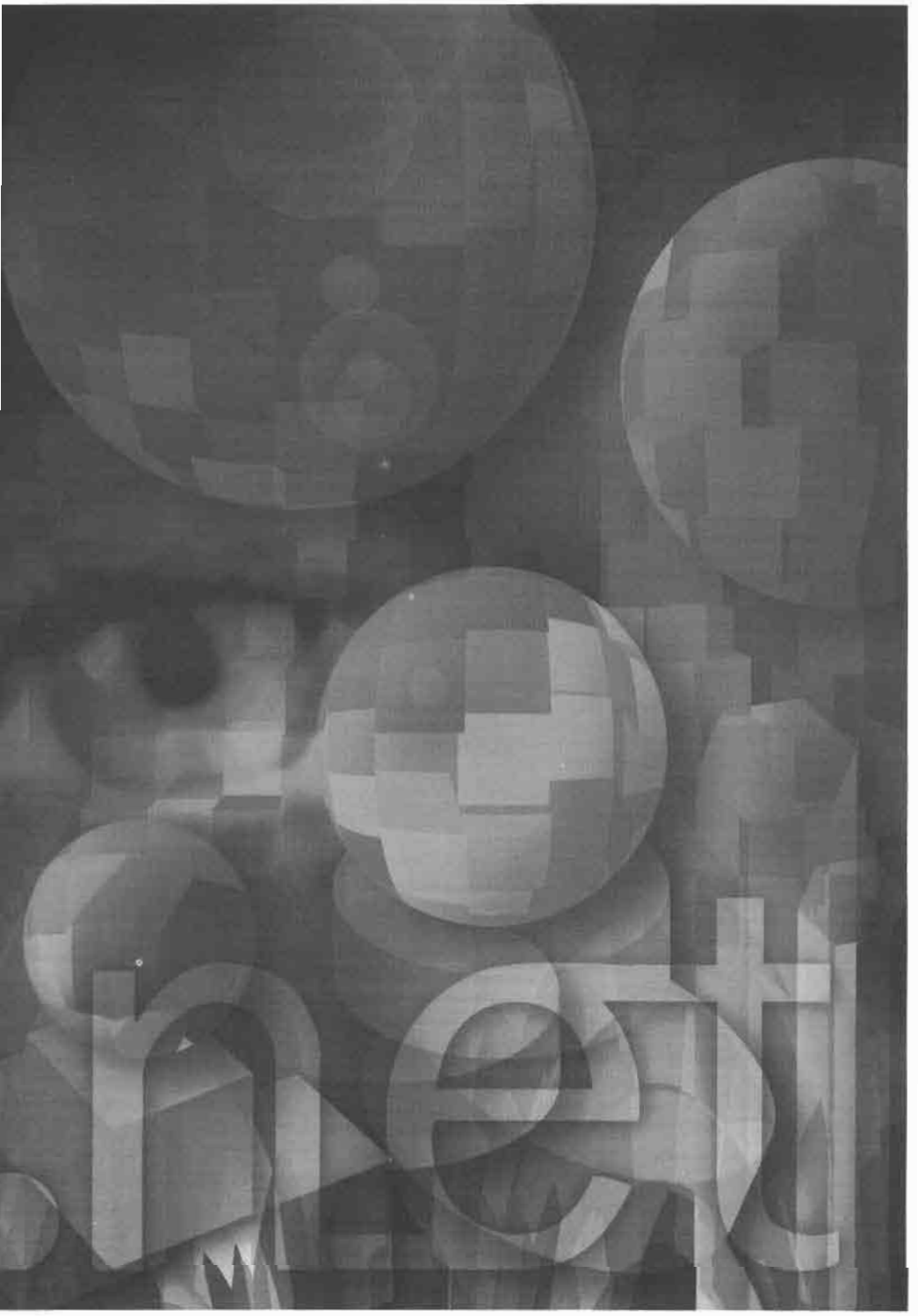
Al redactar este libro el objetivo que tenemos en mente es facilitar al lector un material didáctico que, por sí solo, le permita aprender a programar con Visual Basic .NET y utilizar este lenguaje para desarrollar los tipos de proyecto más habituales. El libro, por lo tanto, no estudia a fondo el entorno de Visual Studio .NET, el funcionamiento de la plataforma .NET o sus servicios, centrándose en las novedades de Visual Basic .NET como lenguaje y el trabajo diario con él.

Comenzando desde el primer capítulo, el que tiene a la vuelta de esta página, podrá seguir procedimientos prácticos para familiarizarse básicamente con el entorno de trabajo y los fundamentos del funcionamiento de la plataforma, centrándonos a partir de entonces en el estudio del lenguaje, el desarrollo de aplicaciones Windows y Web, componentes y servicios.

En los distintos capítulos aprenderá a usar, desde Visual Basic, ADO.NET para trabajar con datos procedentes de bases de datos como SQL Server, GDI+ para crear gráficos, manipular imágenes o imprimir información, ASP.NET para diseñar interfaces de usuario y servicios Web, Crystal Reports.NET con el fin de crear informes, etc.

También aprenderá a crear aplicaciones, sin importar su tipo, usando múltiples hilos de ejecución concurrentes, reutilizando componentes COM heredados de proyectos previos o bien accediendo a los servicios nativos del API de Windows.

Disponiendo de Visual Studio .NET, la beta 2 en castellano o bien la versión definitiva, o de Visual Basic .NET, como producto separado, lo único que necesitará para comenzar a desarrollar sus propias aplicaciones es el libro que tiene en sus manos. Pase la página y póngase cómodo. No pierda el ordenador de vista y prepárese para comenzar.



1
Primera toma de contacto

En este capítulo, asumiendo que ya tiene instalado en su sistema Visual Basic .NET o alguna de las ediciones de Visual Studio .NET, vamos a experimentar nuestra primera toma de contacto con el entorno y, de paso, aprenderemos algunos conceptos básicos a fin de crear nuestra aplicación .NET con Visual Basic .NET. El objetivo principal es ayudarle a perder el miedo durante esos primeros minutos en los que ha de enfrentarse a una nueva herramienta, con un entorno que desconoce, y que suelen ser los más difíciles.

Recuerde que este libro está redactado utilizando una versión preliminar de Visual Studio .NET en castellano, por lo que en su caso, utilizando posiblemente una versión definitiva del mismo producto, puede encontrar diferencias puntuales en las imágenes, títulos de opciones de menú y elementos similares.

Observe que en este capítulo el contenido teórico es mínimo. Para iniciar Visual Basic .NET y crear un programa simple no lo necesitamos. En los capítulos siguientes tendremos tiempo de ocuparnos teóricamente sobre aspectos del lenguaje, el entorno y la plataforma .NET.

Puesta en marcha de Visual Basic .NET

Recurriendo, como es habitual, al menú **Inicio** de Windows, encontrará en la lista de programas una entrada correspondiente a Visual Basic .NET o Visual Studio .NET, según el producto que tenga instalado. Seleccionándola pondrá en marcha el entorno de desarrollo. En la ventana de presentación, la típica

splash que tan sólo permanece unos segundos en pantalla, podrá ver los elementos relacionados con Visual Studio .NET que tiene instalados en su sistema. En la figura 1.1, por ejemplo, puede ver que el entorno usado es la *Beta 2* de *Visual Studio .NET Enterprise*, que cuenta con Visual Basic .NET, Visual C# .NET, Visual C++ .NET, Crystal Reports y Application Center Test.

Figura 1.1. En la ventana de inicio podemos ver los productos que tenemos instalados

Si añadimos al entorno otros productos, como Visual J# .NET, COBOL .NET o Perl .NET, por poner algunos ejemplos, estos aparecerán también en forma de iconos en esa misma ventana de inicio. En cualquier caso, a nosotros nos basta con que exista Visual Basic .NET que, en definitiva, es la herramienta en la que vamos a centrar nuestra atención.

La página de inicio

Tras una instalación nueva de Visual Basic .NET, asumiendo que es la primera vez que iniciamos el entorno, lo primero que vemos aparecer es la **Página de inicio**, concretamente la sección llamada **Mi perfil**. En ella existen varios apartados que nos permiten configurar el comportamiento del entorno para que se adapte lo mejor posible a nuestros conocimientos (véase figura 1.2).

Desplegando la lista titulada **Perfil**, la que aparece en la parte superior de la página, podremos indicar la herramienta que estamos acostumbrados a utilizar: Visual Basic, Visual C++, Visual InterDev, etc. Dependiendo de la opción elegida, automáticamente se establecerá la configuración de teclado y distribución de ventanas de Visual Basic .NET. Estos elementos, no obstante, también pueden establecerse de manera individual. En la figura 1.2, por ejemplo, se ha optado por la distribución de teclado habitual de Visual Basic pero manteniendo la distribución de ventanas predeterminada de Visual Studio .NET.

Otros aspectos que puede configurar en esta sección es el filtro para acceder a la ayuda y si ésta se mostrará en una ventana interna, dentro del mismo entorno, o bien como una ventana independiente. Por último, en la parte inferior,

puede indicar qué desea que ocurra cada vez que inicie de nuevo Visual Basic .NET. Puede dejar que aparezca siempre la **Página de inicio**, concretamente la sección **Introducción**, o bien indicar que se abra el último proyecto con el que se trabajó, que se muestre un cuadro de diálogo para seleccionar el proyecto, etc.

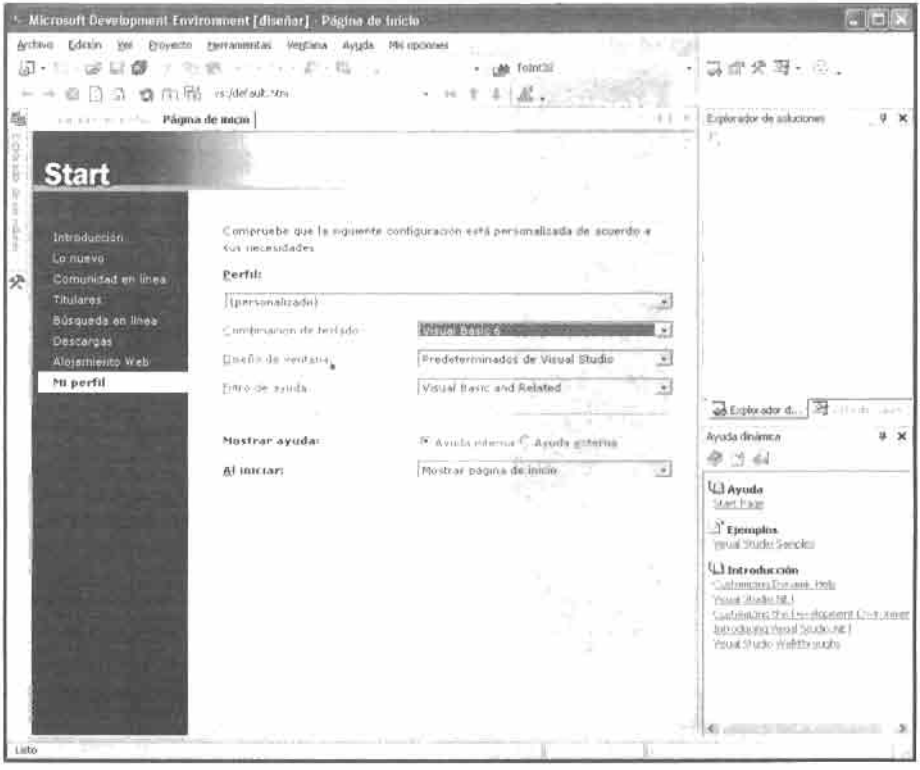


Figura 1.2. Ajustamos nuestro perfil utilizando la página de inicio

Nota

Si en algún momento *pierde* la Página de inicio de vista, porque la haya cerrado inadvertidamente, puede recuperarla pulsando el botón o seleccione la opción Ver>Explorador Web>Inicio.

En caso de que mantenga en la opción **Al iniciar** el valor ofrecido por defecto, cada vez que inicie Visual Basic .NET se encontrará con una ventana como la de la figura 1.3. Realmente es otra sección de la misma **Página de inicio**, en este caso ofreciendo una lista de los últimos proyectos con los que se ha trabajado, para abrirlos con un simple clic, así como los botones necesarios para abrir otros proyectos o iniciar un nuevo.

Utilizando el resto de las secciones de la **Página de inicio** se podrá acceder a la comunidad en línea, a través de Internet, noticias sobre Visual Basic .NET,

actualizaciones, etc. Tan sólo tiene que pulsar cada una de las opciones y experimentar para ver el resultado.

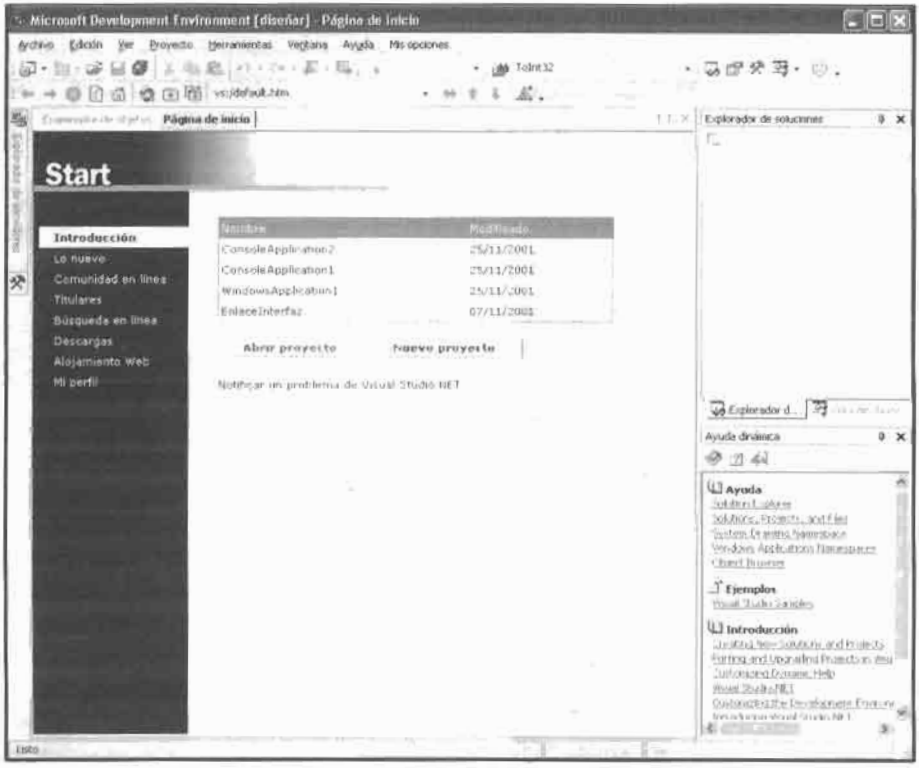


Figura 1.3. Aspecto de la Página de inicio ofreciendo las opciones para abrir e iniciar proyectos

Elementos fundamentales

Además de la **Página de inicio**, en el entorno de Visual Basic .NET encontrará algunos elementos más, algunos de ellos fundamentales para poder desarrollar nuestro trabajo.

En realidad, los elementos visibles dependerán de la distribución de ventanas que previamente, en la sección **Mi perfil**, hayamos seleccionado. Asumiendo que hemos mantenido la disposición por defecto, en la parte derecha podemos ver dos ventanas adosadas. La primera, en la parte superior, cuenta con dos páginas: **Explorador de soluciones** y **Vista de clases**. En la inferior aparece la ventana **Ayuda dinámica**.

El entorno de desarrollo en el que trabaja Visual Basic .NET es capaz de ocultar automáticamente las ventanas que no están utilizándose, mostrándolas de manera inmediata en cuanto el puntero del ratón se sitúa en el lugar adecuado. En la parte superior izquierda del entorno, según la ventana mostrada en la figura 1.3, puede ver que hay una pestaña con un icono y un texto en verti-

cal: Explorador de soluciones. Debajo hay un botón con otro icono. Si sitúa el puntero del ratón sobre él verá aparecer el Cuadro de herramientas.

La parte superior de la ventana principal está ocupada por un menú de opciones y unas paletas de botones. Con ellos podrá iniciar un nuevo proyecto, abrir y guardar archivos, acceder a la Página de inicio, navegar por la ayuda, etc. Conocerá algunos detalles más sobre el entorno en un capítulo posterior. Por ahora simplemente tenemos una noción general de lo que hay alrededor.

Inicio de un nuevo proyecto

Con el fin de cumplir con el habitual ritual que prácticamente exige que el primer programa que se escribe con un nuevo lenguaje o herramienta sea el típico `Hola mundo`, vamos a ver cuáles serían los pasos que daríamos en Visual Basic .NET para conseguir dicho objetivo. De paso veremos cómo crear una aplicación de consola con Visual Basic, una nueva característica inexistente en versiones previas de este producto.

Asumiendo que acaba de iniciar Visual Basic .NET y que, por tanto, tiene la Página de inicio delante, para iniciar un nuevo proyecto tiene las alternativas indicadas en la figura 1.4. Podemos pulsar el botón **Nuevo proyecto**, ya sea el que aparece en la parte inferior de la mencionada Página de inicio o el que hay en la barra de botones, en la parte superior. La otra alternativa es desplegar el menú Archivo y seleccionar la opción Nuevo>Proyecto.

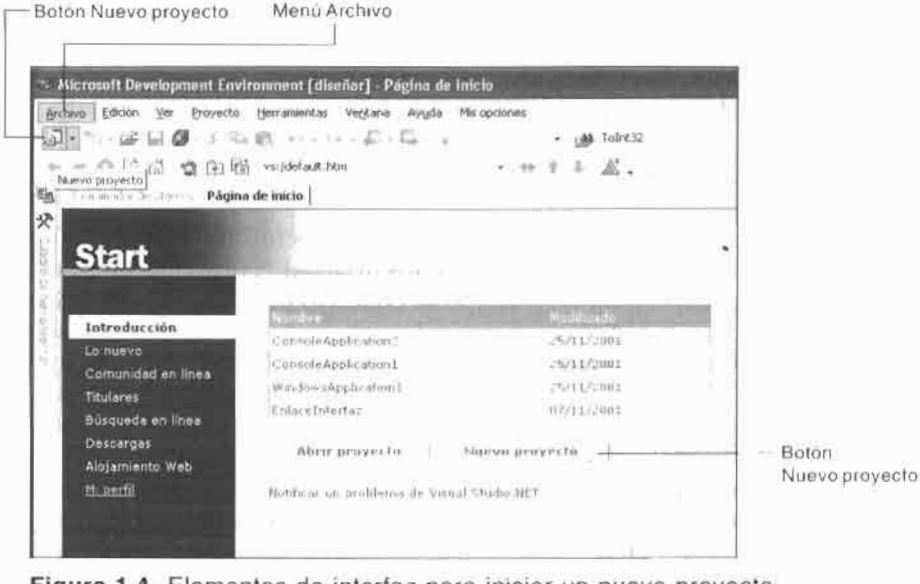


Figura 1.4. Elementos de interfaz para iniciar un nuevo proyecto

En cualquier caso, independientemente del método usado, veremos aparecer el cuadro de diálogo Nuevo proyecto. En él encontrará dos listas, según se

aprecia en la figura 1.5. La de la izquierda contiene varias carpetas agrupando los distintos tipos de proyecto disponibles, mientras que en la de la derecha aparecen los elementos de la carpeta seleccionada a la izquierda. En nuestro caso hemos elegido a la izquierda la carpeta Proyectos de Visual Basic y a la derecha el elemento Aplicación de consola.

Nota

Los tipos de proyecto disponibles en la herramienta dependerán del producto que tengamos instalado. La figura 1.5 corresponde a Visual Studio .NET, de ahí que existan carpetas de proyecto Visual C# y Visual C++. Si tenemos instalado Visual Basic .NET como producto separado, lógicamente esas carpetas no existirán.

Figura 1.5. Cuadro de diálogo Nuevo proyecto

Además de indicar el tipo de proyecto que deseamos iniciar, en este caso una aplicación de consola, en los apartados Nombre y Ubicación deberemos introducir el camino en el que se creará el proyecto y su nombre. Éste determinará el nombre de la carpeta en la que se almacenarán todos los archivos con que cuente el proyecto.

Análisis del código

Introducidos todos los parámetros en el cuadro de diálogo anterior, al pulsar **Intro** o bien el botón equivalente se creará el nuevo proyecto. Sus elementos podremos verlos en la ventana del Explorador de soluciones, mientras que el código correspondiente al único módulo existente en el proyecto aparecerá abierto en el correspondiente editor (véase figura 1.6). Tan sólo tenemos que introducir las sentencias a ejecutar cuando este proyecto se ejecute.

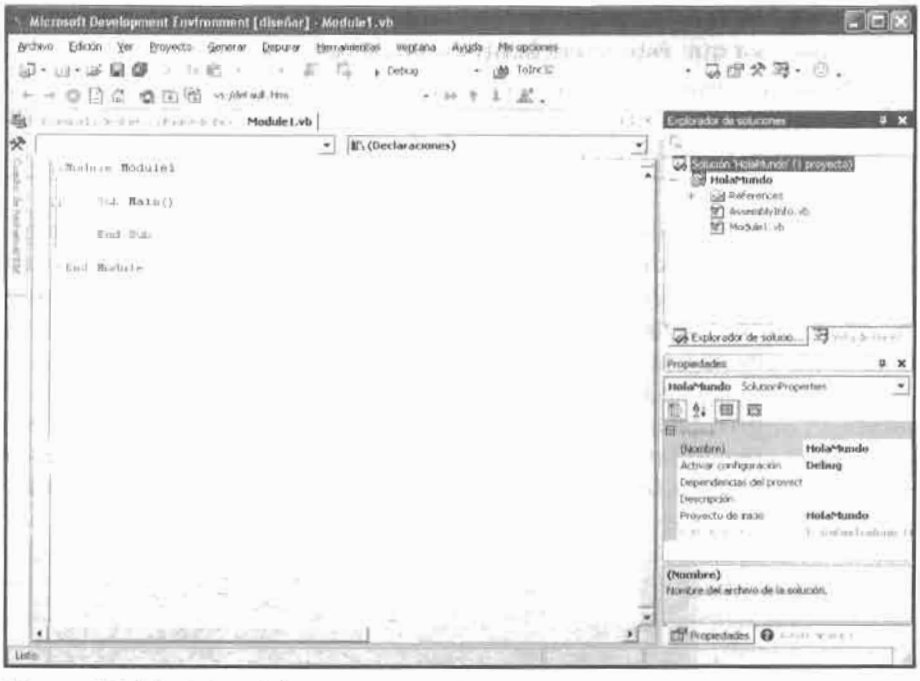


Figura 1.6. El código del nuevo proyecto aparece en el editor de Visual Basic .NET

El código define un módulo, llamado `Module1`, que contiene una función llamada `Main()`. En su interior introducimos la sentencia `System.Console.WriteLine("Hola mundo")`, de tal forma que el programa completo sería el siguiente:

```
Module Module1

  Sub Main()
    System.Console.WriteLine("Hola mundo")
  End Sub

End Module
```

Lo que hacemos en esa sentencia es invocar al método `WriteLine()` de la clase `Console`, una clase definida en el ámbito `System`. Podemos llamar directamente al método, sin crear previamente un objeto de la clase `Console`, porque es un método compartido o estático. Conocerá todos estos detalles en capítulos posteriores, por ahora es suficiente con que tenga una idea global de lo que está haciendo.

El mencionado método `WriteLine()` puede tomar distintos parámetros, entre ellos una cadena de caracteres. Cuando introduzca el código en el editor, al abrir el paréntesis para la lista de parámetros, verá aparecer una pequeña ventana emergente de ayuda indicándole los parámetros necesarios. Observe que dicha ventana cuenta con unos pequeños botones, en forma de flecha, que

nos permiten elegir entre varias versiones diferentes del método `WriteLine()`, ya que éste se encuentra *sobrecargado*.

Ejecución

Para desarrollar este sencillo programa no hemos utilizado ningún diseñador, ni editado propiedades ni trabajado visualmente con componentes. Tan sólo nos hemos servido de un asistente, el que ha generado la mayor parte del código, y usado el editor para introducir una sentencia. El resultado, no obstante, es una aplicación de consola completa.

Basta la pulsación de la combinación de teclas **Control-F5** para compilar el proyecto, obteniendo el correspondiente ejecutable, e iniciarlo. El resultado será similar al mostrado en la figura 1.7. Simplemente aparece en una ventana de consola la cadena de texto entregada como parámetro a `WriteLine()`.

Figura 1.7. Resultado de la ejecución del programa

> **Nota**
>
> La combinación de teclas **Control-F5** equivale a la opción Depurar>Iniciar sin depurar. Lo habitual es usar Depurar>Iniciar, o pulsar **F5**, pero en ese caso la ventana de consola se cerraría de inmediato al finalizar la ejecución del programa, impidiéndonos observar el resultado.

Antes de ejecutarse, el código ha sido compilado generando código en MSIL (*Microsoft Intermediate Language*). Dicho código se almacena en un *ensamblado* que, en apariencia, es como un ejecutable cualquiera. Encontrará dicho ensamblado, en este caso `HolaMundo.exe`, en la subcarpeta `bin` de la carpeta correspondiente al proyecto.

Los proyectos generados con Visual Basic .NET tan sólo pueden ser ejecutados en sistemas donde esté instalada la plataforma Microsoft .NET. Es decir,

no puede tomar el archivo `HolaMundo.exe`, copiarlo a otro ordenador y esperar a que funcione directamente. La plataforma Microsoft .NET se encuentra incluida en Windows .NET y será distribuida como un añadido para las demás versiones de Windows.

Un vistazo al ensamblado

En este caso, a partir del ejemplo anterior, hemos obtenido un ensamblado compuesto de tan sólo un archivo: el propio ejecutable. Como se ha dicho, el ensamblado no contiene código directamente ejecutable sino MSIL, un código intermedio, no es código fuente ni tampoco ejecutable, que es compilado a posteriori en un proceso que conocerá con mayor detalle en un próximo capítulo.

Podemos echar un vistazo al contenido del ensamblado gracias a la existencia de la herramienta externa `ildasm`. Recurra a la línea de comandos usando la opción Visual Studio .NET Command Prompt que aparece en el grupo de programas de Visual Basic .NET. A continuación cambie a la carpeta en la que se encuentra el proyecto de ejemplo anterior e introduzca el comando `ildasm` seguido del nombre del ejecutable, `HolaMundo.exe` en este caso. Verá aparecer una ventana similar a la de la figura 1.8, en la que se ha desplegado la rama `HolaMundo` y el nodo `Module1`.

Figura 1.8. Contenido de nuestro primer ensamblado construido con Visual Basic .NET

El ensamblado, `HolaMundo.exe`, contiene un manifiesto, representado como MANIFEST en la ventana citada antes, y un ámbito llamado `HolaMundo`. En el interior de ese ámbito encontramos la clase `Module1` que, a su vez, cuenta con un constructor y un método compartido o estático llamado `Main()`.

Haga doble clic sobre el método `Main()`. Verá entonces aparecer una nueva ventana con el código MSIL de ese método, un código que, como se aprecia en la figura 1.9, se asemeja bastante al lenguaje ensamblador.

```
// Module1::Main : void()
method public static void Main() cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00
  // Code size       14 (0xe)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hola mundo"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  nop
  IL_000d:  ret
} // end of method Module1::Main
```

Figura 1.9. Código MSIL del método `Main()`

No vamos a entrar en detalles sobre el código MSIL ya que no nos interesa en este momento, pero puede ver que es un código bastante compacto que, en total, no llega a ocupar 16 bytes. El ensamblado completo, que contiene este código e información adicional, tan sólo ocupa 6 kbytes.

Opciones del proyecto

Al explorar el contenido del ensamblado hemos visto que el módulo `Module1` es, en realidad, una clase, si bien con unos atributos específicos que conoceremos posteriormente. Esta clase está contenida en un ámbito o espacio con nombre llamado `HolaMundo`. Está claro que el nombre de ese ámbito se ha obtenido a partir del nombre de proyecto, nombre que establecimos en el asistente en el momento de la creación.

Desde la ventana de propiedades del proyecto es posible modificar éstos y muchos otros parámetros. Para abrir esa ventana pulse con el botón secundario del ratón sobre el proyecto, en la ventana Explorador de soluciones, y luego seleccione la opción Propiedades (véase la figura 1.10). Aparecerá una ventana como la de la figura 1.11, estando abierta por defecto la página General. En ella se indica el nombre que tendrá el ensamblado, el tipo de aplicación resultante y el nombre del espacio de nombres o ámbito que actuará como raíz de todo el proyecto.

Al ocuparnos del lenguaje Visual Basic .NET propiamente dicho profundizaremos sobre la utilidad y uso de los ámbitos con nombre, una de las novedades incluidas en esta versión.

Figura 1.10. El menú emergente del proyecto nos da acceso a la ventana de propiedades

Figura 1.11. La ventana de propiedades del proyecto cuenta con múltiples páginas de opciones

Puntos clave

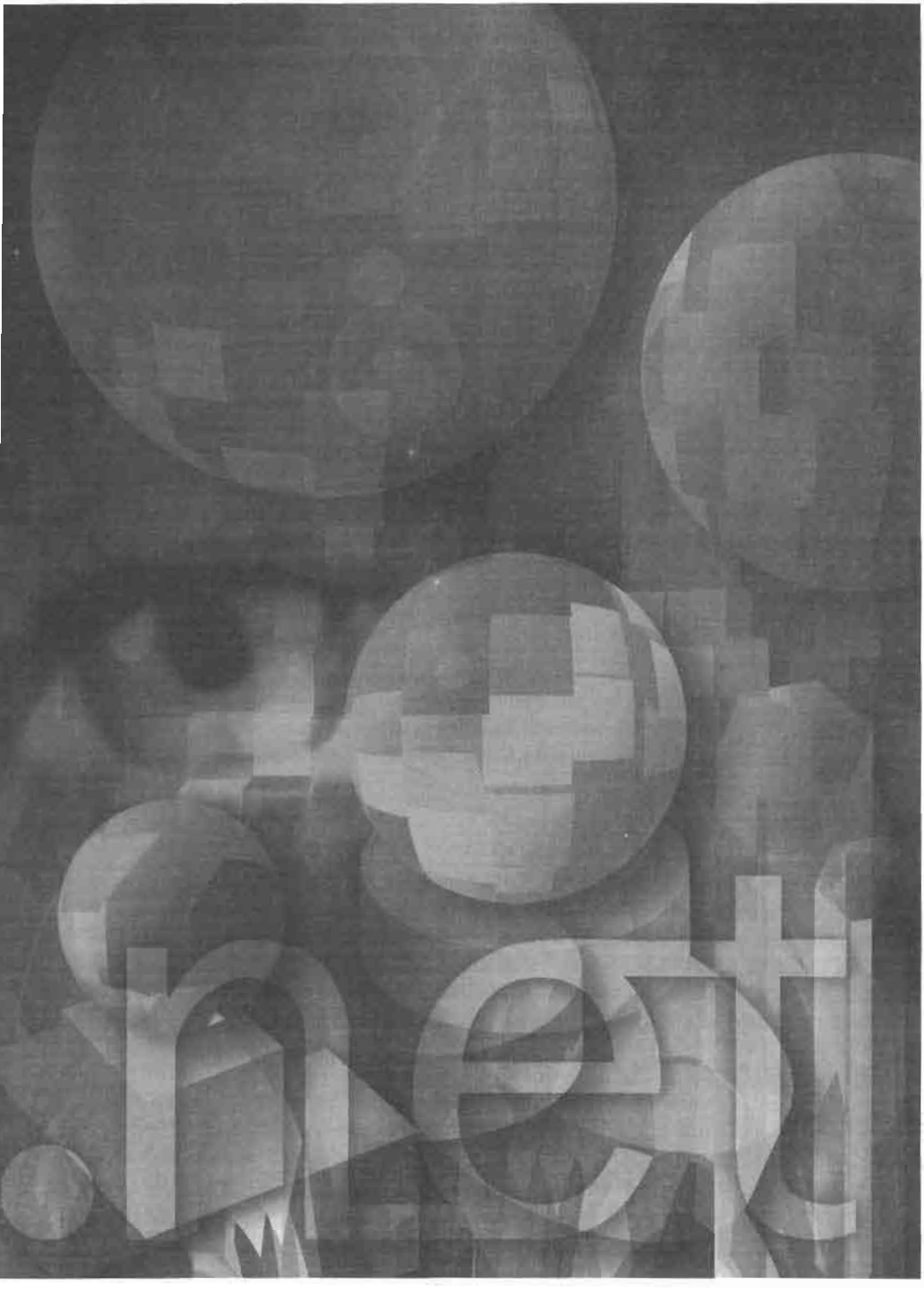
- Al iniciar Visual Basic .NET se encontrará con el entorno de desarrollo de Visual Studio .NET, un entorno que estará compartido con otros lenguajes y herramientas en el caso de que no esté utilizando Visual Basic .NET, como producto independiente, sino algunas de las ediciones de Visual Studio .NET.

- El entorno de desarrollo puede adecuarse a nuestros conocimientos. Utilizando la sección **Mi perfil** de la **Página de inicio** podemos establecer la distribución de teclado, disposición de la ventana y otros parámetros relacionados.
- Cada vez que iniciemos Visual Basic .NET nos encontraremos con la **Página de inicio**, desde la que podemos abrir un proyecto existente o crear uno nuevo.
- En el entorno de desarrollo encontramos algunos elementos fundamentales, como el **Explorador de soluciones**, la ventana de **Ayuda dinámica** o el **Cuadro de herramientas**.
- Desde el **Explorador de soluciones** podemos acceder a los distintos elementos que forman parte de nuestro proyecto, utilizando el menú emergente asociado para manipular esos elementos.
- En la ventana **Nuevo proyecto** se encuentran los asistentes que usaremos para iniciar los distintos tipos de proyecto que pueden desarrollarse con Visual Basic .NET.
- El resultado de un proyecto Visual Basic .NET es un ensamblado. Éste se puede componer de uno o más módulos, en cuyo interior se aloja el código MSIL.
- Los ejecutables obtenidos con Visual Basic .NET no pueden utilizarse directamente en cualquier equipo, es necesario instalar previamente la plataforma Microsoft .NET.
- Utilizando la ventana de propiedades del proyecto podemos configurar multitud de opciones, entre ellas el nombre del ensamblado y el del ámbito en el que se incluirán todos los elementos del proyecto.

Resumen

Este capítulo nos ha servido para acercarnos por primera vez a Visual Basic .NET, principalmente el entorno de desarrollo, siguiendo los pasos necesarios para desarrollar una sencilla aplicación de consola. De manera análoga podríamos haber creado una aplicación Windows con un formulario, una aplicación Web o un componente. El proceso no habría sido mucho más complejo, aunque sí los recursos utilizados por el proyecto.

Lo importante es que, básicamente, ya sabe qué pasos debe dar para iniciar Visual Basic .NET, crear un proyecto, compilarlo y ejecutarlo desde el propio entorno. Ha usado un método de una clase ya existente y utilizado una herramienta para examinar el contenido de un ensamblado. En conjunto, todo ello tiene como objetivo hacerle perder un poco el miedo inicial al comenzar a usar una nueva herramienta. En la práctica, tiene una base inicial para comenzar a trabajar en los próximos capítulos.


.net

2
El entorno de desarrollo

Los programadores somos usuarios de aplicaciones, en nuestro caso herramientas de desarrollo, que, básicamente, cuentan con los mismos elementos que podemos encontrar en cualquier otra aplicación: ventanas con menús de opciones, barras de botones, menús emergentes, paletas de herramientas, etc. En este sentido, el entorno de desarrollo de Visual Basic .NET no se diferencia en exceso del que podríamos encontrar en un programa de retoque fotográfico, una hoja de cálculo o una aplicación de diseño. La diferencia, obvia, es que nosotros tratamos con elementos relativamente abstractos: diseñamos ventanas y componentes y codificamos una lógica de implementación que queda oculta.

Para poder hacer nuestro trabajo necesitamos, lógicamente, conocer el entorno en el que vamos a desenvolvernos. En nuestro caso esto significa conocer los elementos del entorno de desarrollo de Visual Basic .NET. Ése es el objetivo de este capítulo: indicarle qué elementos son los que puede encontrar y su utilidad. No se trata de establecer un material de referencia, para eso ya está la propia ayuda del entorno, sino de descubrir rápidamente los elementos más importantes y la manera de utilizarlos.

Disposición general

El entorno de desarrollo de Visual Basic .NET usa por defecto un esquema de ventanas adosadas o ancladas unas a otras. Existe un elemento central: la ventana en la que aparece la **Página de inicio**, los distintos editores de código y

diseñadores visuales. Esa parte ocupa, como acaba de decirse, el espacio central. Los distintos elementos abiertos aparecen como páginas apiladas unas sobre otras, con una pestaña en la parte superior que facilita su selección.

Aparte de ventanas adosadas unas a otras y ventanas apiladas en forma de páginas, también podemos encontrar ventanas que se ocultan automáticamente, cuando no son necesarias, y aparecen al situar el puntero del ratón sobre la pestaña que contiene su título. Las ventanas también pueden quedar como flotantes, sin ningún enlace visual con el entorno. En cualquier caso, todos estos estados pueden alterarse con una simple operación de arrastrar y soltar o un sencillo clic de ratón.

En la figura 2.1 se indica el estado de algunas ventanas. La parte central está ocupada por un diseñador de esquemas XML, en la parte superior, y el editor de código, en la inferior. A la derecha aparecen la ventana **Ayuda dinámica** y la ventana **Propiedades**, adosadas una sobre la otra. En el margen izquierdo del entorno, verticalmente, podemos ver dos pestañas que corresponden al **Explorador de servidores** y el **Cuadro de herramientas**.

Para controlar la ocultación automática de las ventanas se utiliza el botón que aparece en el extremo derecho de la barra de título, junto al botón de cierre. Dicho botón simula una chincheta que podemos pinchar, para que la ventana quede fija en su posición, o bien soltar, para que se oculte automáticamente cuando no esté siendo necesaria. El botón de ocultación automática sólo aparece en las ventanas que están adosadas a algún margen del entorno, no en las ventanas flotantes.

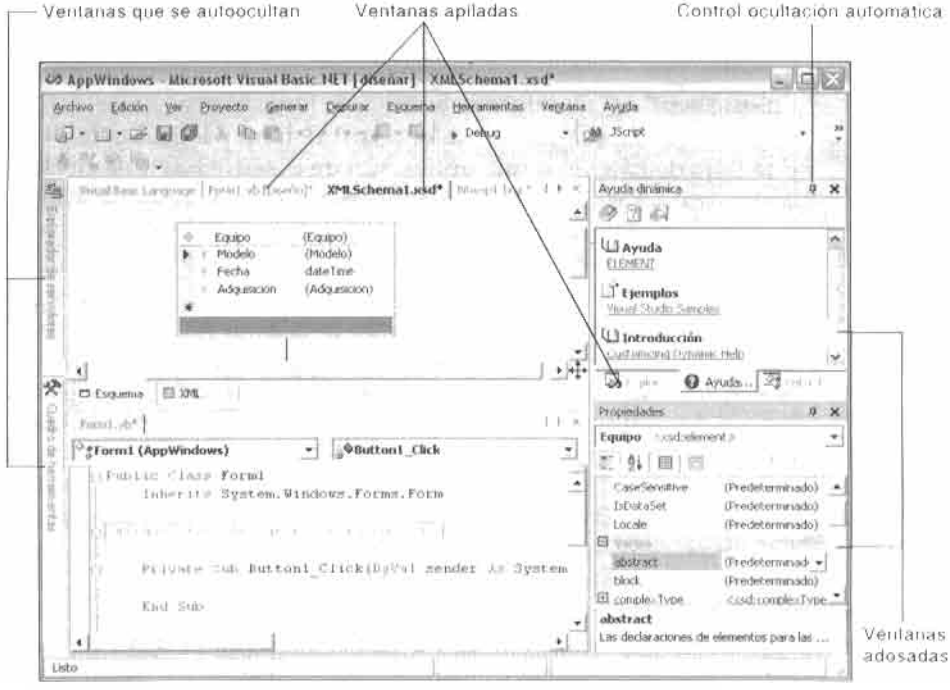


Figura 2.1. Estado de las ventanas que hay en el entorno

Una ventana que no se oculta automáticamente permanece siempre adosada a un margen de la ventana principal de Visual Basic .NET, no siendo posible modificar este estado hasta en tanto no desactivemos la propiedad de ocultación automática. En ese momento, podemos tomar la ventana, con el botón principal del ratón, y arrastrarla por el entorno hasta una nueva localización. Si tomamos la ventana pulsando sobre su barra de título desplazaremos todo el conjunto, en caso de que haya múltiples ventanas apiladas en páginas. Si tan sólo quiere operar sobre una de ellas, en lugar de la barra de título pulse sobre la pestaña que representa a esa ventana.

A medida que vaya arrastrando la ventana sobre el entorno, un borde dinámico le irá mostrando cómo quedaría si la soltase en ese momento. Pruebe a acercar el puntero del ratón a los márgenes de la ventana principal, los de otras ventanas ya adosadas o flotantes o la barra de título de esas ventanas. La ventana arrastrada quedará adosada al margen o apilada como una nueva página, según los casos. Si libera la ventana lejos de cualquier margen conseguirá que quede como flotante.

Las páginas que ocupan el área central del entorno, en la que aparecen diseñadores, editores y el navegador Web, aunque pueden adosarse unas a otras, en lugar de aparecer todas apiladas en forma de páginas, no pueden quedar flotantes. Recurra a las opciones del menú emergente, pulsando el botón secundario del ratón sobre cualquiera de las pestañas, para conseguir un efecto idéntico sin necesidad de arrastrar y soltar.

Nota

Usando la ventana de opciones del entorno, que conocerá posteriormente, es posible cambiar el mecanismo de funcionamiento de las ventanas de Visual Basic .NET de tal manera que en lugar de aparecer ancladas unas a otras, que es la opción por defecto, se ajusten al funcionamiento típico de una aplicación MDI (*Multiple Document Interface*), con una ventana principal conteniendo las ventanas hija de editores y diseñadores.

Soluciones y proyectos

Los distintos elementos que se usan en Visual Basic .NET para desarrollar aplicaciones: formularios, recursos, código, etc., se agrupan de manera jerárquica en varios niveles. La raíz, o el nodo principal, es la *solución*, que puede contener uno o más proyectos. En cada proyecto existirán formularios, controles, definiciones de clases, referencias externas, etc. Mediante el Explorador de soluciones podemos gestionar todos esos elementos.

El Explorador de soluciones aparece inicialmente, asumiendo que estamos utilizando la configuración por defecto de Visual Basic .NET, en la parte superior derecha del entorno. Si no lo ve, puede hacerlo aparecer mediante la combinación **Control-Alt-L**. También puede usar la opción Explorador de soluciones

del menú Ver consiguiendo el mismo resultado. En la figura 2.2 se puede ver esta ventana con los elementos de una aplicación Windows simple. Los menús emergentes son los que corresponden al proyecto.

Como puede ver, la solución aparece como primer elemento de la ventana, en este caso contando con tan sólo un nodo: el proyecto `AppWindows`. Este dispone de una carpeta de referencias y dos módulos. El llamado `Form1.vb` corresponde al formulario existente en el proyecto. Haciendo doble clic sobre cualquiera de estos elementos lo abriremos en su respectivo editor o diseñador. También puede usar los dos primeros botones que hay en la parte superior de la ventana, contando de izquierda a derecha, para acceder al código o al diseñador del elemento seleccionado en ese momento. Usando el menú emergente asociado a cada elemento es posible realizar diversas tareas. Si pulsa el botón secundario del ratón sobre un proyecto, por ejemplo, podrá generarlo, eliminarlo, cambiar el nombre, añadirle nuevos elementos o nuevas referencias.

Cada uno de los elementos que aparece en el Explorador de soluciones se almacena en un archivo con una determinada extensión y formato. La solución, que relaciona a los demás elementos, se aloja en un archivo de texto con extensión `sln`. Los proyectos, dependiendo del lenguaje, se almacenan en archivos con extensiones diversas. En el caso de Visual Basic la extensión es `vbproj`, siendo su contenido XML. Algo similar ocurre con los módulos de código y correspondientes a formularios, cada uno de los cuales se escribe en un archivo con extensión identificadora del lenguaje.

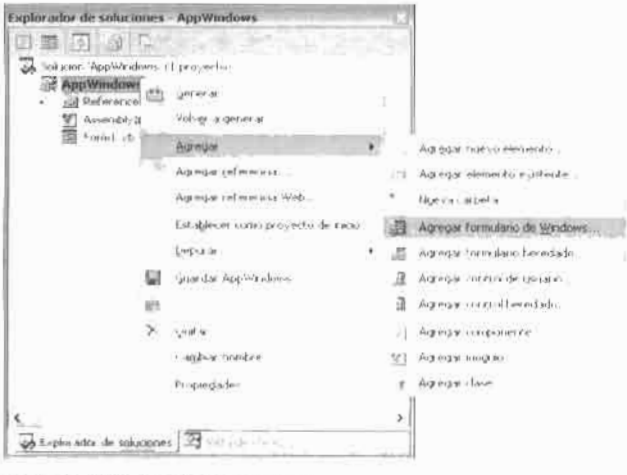


Figura 2.2. Ventana del Explorador de soluciones

Nota

Eche un vistazo a la carpeta del proyecto creado en el capítulo previo para ver los distintos archivos generados. También puede seleccionar un elemento cualquiera en el Explorador de soluciones y ver el camino y nombre del archivo correspondiente en la ventana Propiedades.

Objetos existentes en un módulo

Las soluciones, los proyectos y las carpetas actúan como contenedores que permiten clasificar los módulos de último nivel. Estos contienen definiciones de clases, por ejemplo asociadas a un formulario, un componente o un servicio. Este código siempre podemos verlo en el editor de texto. En algunos casos también es posible utilizar un diseñador para modificar indirectamente ese código, por ejemplo cuando el módulo corresponde a un formulario.

En un módulo pueden existir ámbitos con nombre, definiciones de clases, propiedades, métodos y eventos, entre otros elementos. Mediante la ventana Vista de clases podemos obtener un esquema jerárquico de todos esos elementos, lo cual simplifica su localización y manipulación. Esta ventana, que puede observar en la figura 2.3, se hace aparecer mediante la combinación de teclas **Control-Mayús-C** o la opción Ver>Vista de clases.

Los elementos que aparecen en la figura 2.3 son los que existen en el proyecto `HolaMundo` desarrollado en el capítulo previo. Existe un ámbito con nombre, llamado precisamente `HolaMundo`, en cuyo interior está el módulo `Module1`. Este, a su vez, cuenta con un método: `Main()`. Si el proyecto fuese más complejo, por ejemplo una aplicación Windows con formularios, podríamos acceder con esta ventana a sus propiedades y métodos asociados a eventos.

Utilizando las opciones del menú emergente asociado a cada uno de estos elementos, que haremos aparecer con el botón secundario del ratón, podremos añadir nuevas clases al proyecto, nuevos miembros a las clases, acceder a la definición del elemento, mostrarla en el Examinador de objetos, etc. Son opciones que, en su mayoría, se encuentran en el menú principal del entorno, pero que resulta mucho más cómodo usar desde el menú emergente.

Figura 2.3. Ventana Vista de clases

Nota

En la esquina superior izquierda de la ventana Vista de clases hay un botón *que, al ser pulsado, despliega un menú de opciones. Con ellas puede alterar* el orden en que aparecen los elementos en la lista jerárquica.

Examinando clases y otros elementos

Una de las ventanas que no se encuentra visible inicialmente en el entorno es el Examinador de objetos, que puede resultar muy útil para obtener información diversa sobre los elementos que forman parte de la aplicación y también los que tenemos a nuestra disposición por ser servicios de la plataforma Microsoft .NET.

La combinación de teclas a usar en este caso es **Control-Alt-J**. También puede utilizar la opción Ver>Otras ventanas>Examinador de objetos para mostrar la ventana que se ve en la figura 2.4 ocupando prácticamente todo el espacio disponible en el entorno.

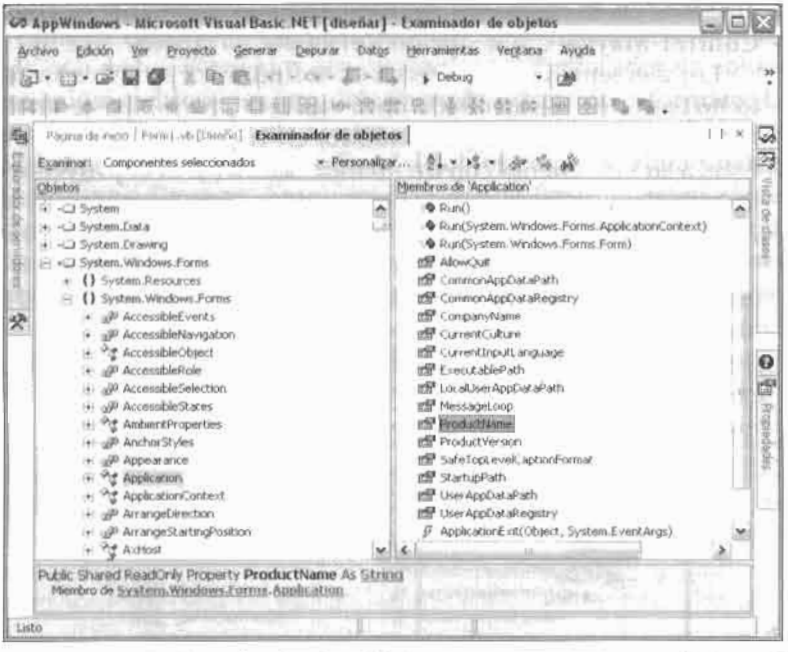


Figura 2.4. El Examinador de objetos aparece como una página más, al mismo nivel que diseñadores y editores

Como se aprecia, la ventana cuenta con dos paneles claramente diferenciados. A la izquierda tenemos la lista de objetos, mientras que a la derecha está la lista de miembros.

Aparte de poder ver los elementos que forman parte de la aplicación y sus miembros, como en la ventana Vista de clases, en el Examinador de objetos podemos acceder también a otros ámbitos y clases.

En la figura 2.4, por ejemplo, se está examinando la lista de miembros de la clase `Application`, definida en el ámbito `System.Windows.Forms`. El miembro seleccionado, en la lista de la derecha, es `ProductName` y, según se indica en la parte inferior, es una propiedad de tipo `String` que puede leerse y modificarse.

Acceso a los asistentes

El Explorador de soluciones y la ventana Vista de clases nos permiten conocer y manipular los elementos que existan actualmente abiertos en el entorno: proyectos, formularios, clases, métodos, etc. Esos elementos son creados, por regla general, mediante el uso de diversos asistentes. Al pulsar el botón **Nuevo proyecto**, o seleccionar ciertas opciones de los menús emergentes de las citadas ventanas, estamos abriendo esos asistentes.

La mayoría de las opciones que hay en la ventana Nuevo proyecto son asistentes que generan un nuevo proyecto, ya sea añadiéndolo a la solución actual o bien creando una nueva solución. Estos asistentes, por regla general, no necesitan ninguna información adicional aparte del nombre del proyecto, porque son lo suficientemente específicos como para saber qué elementos tienen que añadir al proyecto. En algunos casos, no obstante, sí puede ser necesaria esa intervención. Es lo que ocurre al crear un nuevo proyecto de base de dato o de instalación. En el primer caso podremos elegir una referencia a una base de datos, o bien crearla en ese momento, en la ventana de la figura 2.5. En el segundo, al crear un proyecto de instalación, aparece un asistente de varios pasos (véase figura 2.6) que nos guía en el proceso que finalmente añadirá los elementos que se ajusten a nuestras preferencias. En los menús emergentes de muchas ventanas, como Vista de clases o Explorador de soluciones, existen opciones que dan paso a asistentes para añadir nuevos elementos generando automáticamente el código, o un esqueleto, basándose en los datos que introduzcamos.

Figura 2.5. Asistente para agregar una referencia a una base de datos

Desde el Explorador de soluciones, teniendo seleccionado un proyecto, puede usar las opciones del menú emergente para añadir un nuevo formulario, una nueva clase, un nuevo componente, etc. En todos los casos se crea un nuevo módulo de código correspondiente al elemento añadido. Son opciones que también puede encontrar en el menú Proyecto de la barra principal de menús. Tanto en el menú emergente como en el citado menú Proyecto existe una opción, llamada Agregar nuevo elemento, que da paso a la ventana mostrada en la figura 2.7. En ella aparecen, clasificados por tipos, todos los elementos que

es posible añadir al proyecto. Algunas opciones simplemente añaden un archivo al proyecto mientras que otras, como el Asistente para formularios de datos, realizan tareas considerablemente más complejas, conectando con una base de datos, definiendo el conjunto de columnas y filas sobre las que va a operarse y diseñando una interfaz de usuario para ello.

Además de asistentes, en el menú de opciones de Visual Studio.NET encontraremos múltiples herramientas mediante las que añadir referencias al proyecto actual, generar páginas HTML con información sobre dicho proyecto, etc.

Figura 2.6. Asistente para crear un proyecto de instalación

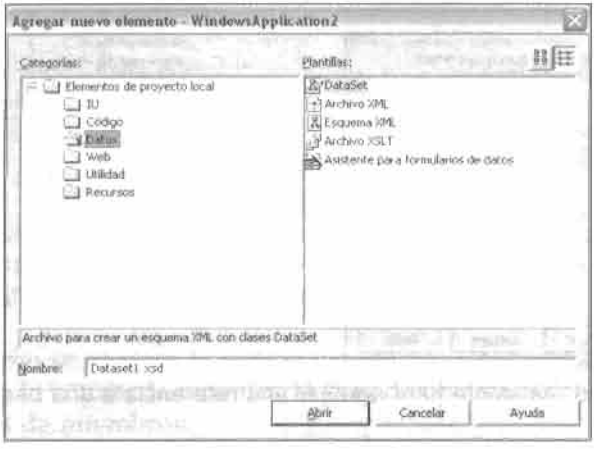


Figura 2.7. Opciones de la ventana Agregar nuevo elemento

Diseñadores y editores

Las clases, los formularios, las páginas HTML, los archivos XML y, en general, todos los elementos que se generan a partir de asistentes y opciones de Vi-

sual Basic .NET deben ser personalizados, añadiendo los objetos precisos para que la aplicación final desarrolle la función que se espera de ella. Con este fin utilizaremos diseñadores y editores diversos, así como los elementos que aparecen en el Cuadro de herramientas.

Desde el Explorador de soluciones es posible acceder al diseñador y/o editor del elemento que se tenga seleccionado en ese momento. Con este fin se utilizan los botones reseñados en la figura 2.8. El segundo, que se usa para abrir el diseñador que corresponda, sólo aparece en caso de que el elemento seleccionado tenga asociado un diseñador.

El editor utilizado es siempre el mismo, si bien su funcionalidad se adapta al tipo de archivo sobre el que se esté trabajando. Los diseñadores, por el contrario, son específicos y facilitan tareas tan diversas como el diseño de una interfaz de usuario Windows o Web, el diseño de un esquema XML, etc.

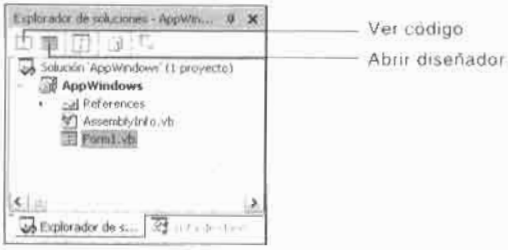


Figura 2.8. Acceso al editor o diseñador asociado a un elemento del proyecto

Diseño de formularios

Hasta ahora no ha utilizado ningún diseñador de Visual Basic .NET y, como supondrá, uno de los más utilizados es el que nos permite diseñar formularios para aplicaciones Windows. Su funcionamiento es similar al usado para crear interfaces Web ya que, fundamentalmente, su finalidad y comportamiento son prácticamente idénticos. En el diseñador de formularios Windows el contenedor es una ventana, mientras que en el de formularios Web dicho contenedor es una página HTML.

Inicie un nuevo proyecto seleccionando el elemento Aplicación para Windows de la ventana Nuevo proyecto. Tras un momento se encontrará con un formulario, con el título Form1 en la parte superior, y una importante lista de controles en el Cuadro de herramientas. Puede seleccionar algunos controles, pulsando sobre ellos con el botón principal del ratón, e insertarlos en el formulario efectuando la misma operación en la posición donde desee colocarlos. Lógicamente tiene que hacerlo uno a uno, es decir, no puede seleccionar simultáneamente varios controles.

Además de componentes visuales, como las etiquetas de texto, las cajas de texto o los botones, existen otros, por el contrario, que son componentes no visuales. Si inserta en un formulario un componente `ToolTip` o un `ImageList`, por poner dos ejemplos, verá que el diseñador los muestra aparte, en la parte inferior (véase la figura 2.9), pero no en el interior del formulario.

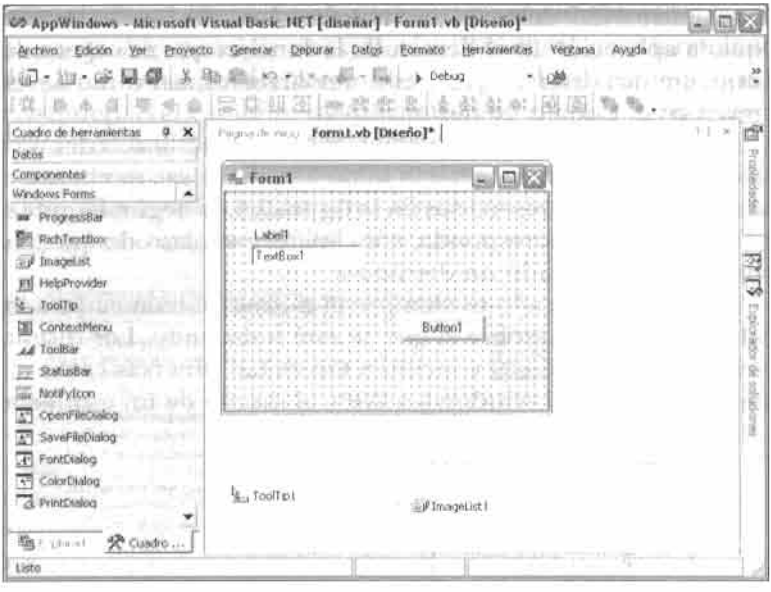


Figura 2.9. El diseñador muestra los componentes dentro o fuera del formulario según su tipo

A medida que vamos insertando componentes en el contenedor, independientemente de dónde aparezcan, el diseñador se encargará de actualizar adecuadamente el código. Al insertar un botón, por poner un ejemplo, el diseñador generará el código Visual Basic o HTML para completar la definición de la clase o página, según el caso, con la declaración de ese botón. De manera análoga, al modificar sus dimensiones o posición, es el diseñador quien actualiza dicho código.

Mediante el diseñador de formularios podemos mover los componentes, cambiar su tamaño y acceder a las opciones de su menú emergente. Por sí solo, no obstante, nos sería de poca utilidad. Para completar su trabajo son fundamentales las ventanas Propiedades y Cuadro de herramientas.

Edición de propiedades

Como acaba de decirse, uno de los elementos indispensables al trabajar con un diseñador es la ventana Propiedades. En realidad esta ventana tiene más funciones, ya que puede utilizarse también para acceder a las propiedades de cualquier elemento que haya seleccionado en el Explorador de soluciones, a pesar de que no haya abierto un diseñador.

La ventana Propiedades es bastante similar a la que encontramos en Visual Basic 6.0. En la figura 2.10 puede ver indicados algunos de los elementos de la mencionada ventana, particularmente la finalidad de cada uno de los botones que aparecen en su parte superior.

La lista de propiedades puede aparecer ordenada por categorías, es la opción por defecto, o bien alfabéticamente. Se altera este orden con los dos pri

meros botones. El siguiente botón se utiliza para mostrar la lista de propiedades, que es el estado por defecto. En la parte inferior aparece una pequeña descripción de la propiedad seleccionada en ese momento, correspondiente al componente que se muestra en la lista desplegable que hay justo debajo del título de la ventana.

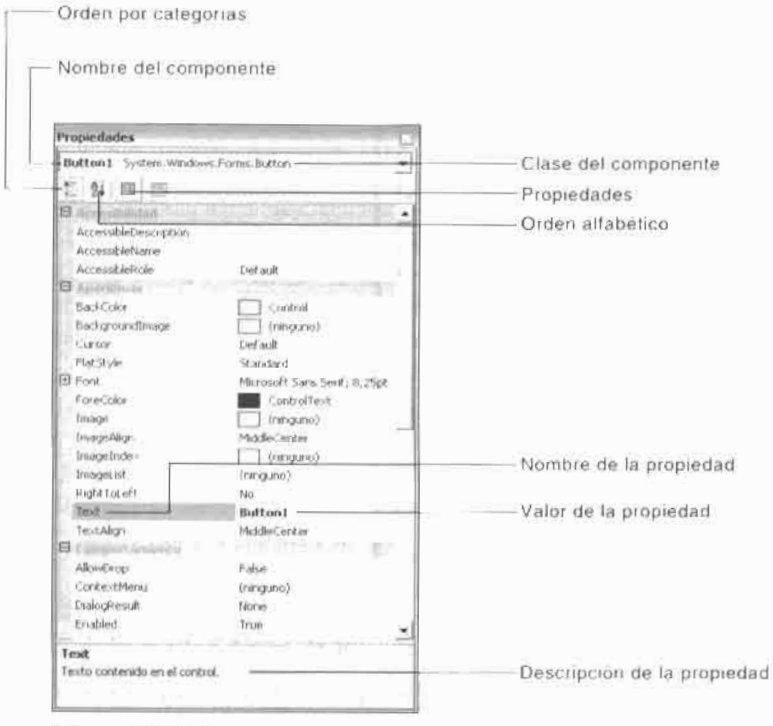


Figura 2.10. Elementos de la ventana Propiedades

Cada entrada de la lista que ocupa la parte central de la ventana se compone de dos elementos: el nombre de la propiedad y su valor actual. El valor de una propiedad depende de su tipo, puede ser una cadena de caracteres, un número, un valor de una enumeración, un color o incluso un objeto.

Las propiedades y los métodos son miembros de un objeto, concretamente del componente seleccionado. En un capítulo posterior aprenderá a definir este tipo de miembros. Cada miembro, propiedad o evento, dispone de un nombre estático, que no podemos alterar, y un valor que suele ser dinámico y podemos modificar de distintas formas, dependiendo del tipo de la propiedad. Propiedades como `Text` o `Name`, por ejemplo, almacenan una cadena de caracteres, un valor que, en la ventana **Propiedades**, se modifica utilizando el teclado, como haríamos en un editor cualquiera.

Otras propiedades, por el contrario, muestran directamente una lista de los valores que pueden contener o, incluso, un editor específico. Observe la figura 2.11, en la cual aparece, en la parte superior, la lista de valores que puede tomar la propiedad `DialogResult`. En este caso no es necesario introducir el

valor utilizando el teclado, basta con desplegar la lista y seleccionar uno de los disponibles. Más abajo puede ver el editor de la propiedad Dock. Dicha propiedad determina si un componente se adosará en una cierta posición de su contenedor y, en lugar de seleccionar un valor, el editor muestra gráficamente las posibilidades disponibles.

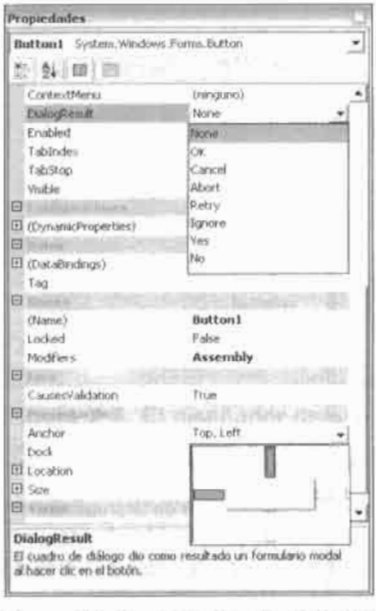


Figura 2.11. Edición de dos propiedades distintas

Cada evento de un componente puede tener asociado el nombre de un método que, como ya sabe, será ejecutando cuando el suceso representado por el evento tenga lugar. La lista de eventos de un cierto objeto aparece en la lista desplegable que hay en la parte superior derecha del editor de código, en lugar de aparecer como una segunda página de la ventana Propiedades. Esto diferencia a Visual Basic .NET de otros lenguajes del entorno que, como C#, si usan esa misma ventana. En la figura 2.12 puede ver la lista de eventos del botón que hay en el formulario. El mecanismo es idéntico al de Visual Basic 6.0.

Para crear un nuevo método asociado a un evento, primero seleccione en la lista de la izquierda el componente que desee y, a continuación, en la lista de la derecha el evento. En ese momento se completará la definición de la clase con un nuevo método, situándose el cursor en el punto adecuado para introducir la implementación.

Componentes y controles

Tanto la ventana Propiedades como el propio diseñador de formularios se usan para modificar objetos o componentes, elementos que podemos encontrar, en su mayor parte, en la ventana Cuadro de herramientas. El Cuadro de

herramientas dispone de múltiples páginas, cada una de las cuales contiene un grupo de componentes. Existen componentes genéricos, que pueden utilizarse indistintamente del tipo de proyecto que esté desarrollándose, y otros mucho más específicos. Los componentes de acceso a datos, por ejemplo, son del primer tipo y cuentan con su propia página, mientras que los usados en formularios Windows o Web son específicos del tipo de aplicación.

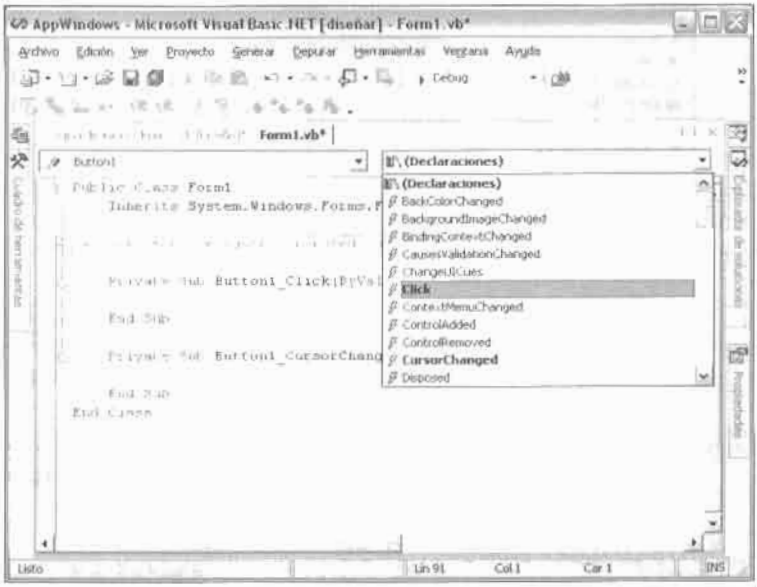


Figura 2.12. Edición de los eventos de un componente

Mediante el menú emergente asociado al Cuadro de herramientas podemos, como se aprecia en la figura 2.13, modificar el orden en que aparecen los elementos, cambiar el modo de visualización entre Lista, el modo por defecto, y una vista simple de iconos, sin nombres de componentes. También es posible modificar los nombres de los componentes, aunque no es algo habitual en los componentes originales, y añadir más fichas o páginas a esta ventana. En la parte inferior puede ver una ficha llamada Mis trozos de código, añadida con el fin de alojar en ella porciones de código a usar en múltiples proyectos.

Para cambiar de una página a otra tan sólo hay que pulsar en los títulos que, a modo de botones, pueden verse en la parte superior e inferior, encima y debajo, respectivamente, de la lista de componentes que ocupa la parte central. Si ésta contiene más elementos de los que es posible mostrar de una sola vez, también verá aparecer en el extremo derecho, arriba y abajo, sendos botones en forma de flecha que facilitan el desplazamiento.

Aparte de tomar componentes y controles del Cuadro de herramientas para colocarlos en el interior de un contenedor, este elemento también puede usarse en sentido inverso. Teniendo marcado en el editor cualquier porción de código, mediante la técnica de arrastrar y soltar podemos alojarlo en la página que nos interese del Cuadro de herramientas y darle un nombre. Posteriormente,

cuando lo necesitemos, podemos recuperar esa porción de código de igual forma. Podríamos decir que el Cuadro de herramientas puede utilizarse como un portapapeles persistente, que no pierde su contenido al desconectar y volver a iniciar, y clasificable.

Figura 2.13. El Cuadro de herramientas y su menú emergente

Diseñadores adicionales

Aparte del diseñador de formularios Windows y formularios Web, Visual Basic .NET dispone de otros muchos que facilitan la creación de esquemas XML, páginas HTML, la estructura de conjuntos de datos, imágenes de mapas de bits, iconos y cursores, etc. Cada uno de ellos tiene sus particularidades, si bien hay similitudes, por ejemplo, entre el diseñador que se utiliza para los esquemas XML y el usado para la estructura de conjuntos de datos, o entre el diseñador de mapas bits y el de iconos.

En la figura 2.14 puede ver el diseñador de esquemas XML mostrando un esquema recién creado. Todas las opciones necesarias para añadir elementos, atributos, claves y relaciones se encuentran en el menú emergente asociado a cada objeto de los que aparece en el diseñador. Usando el botón que hay en la parte interior del diseñador podemos acceder al código XML, ya en el editor que revisaremos en el punto siguiente.

El editor de páginas HTML es similar al de formularios Web, si bien los componentes que podemos incluir, alojados en una página distinta del Cuadro de herramientas, no son componentes ASP.NET sino componentes HTML puros.

Mediante el editor que aparece en la figura 2.15 se facilita la edición de cualquier tipo de imagen que pudiéramos necesitar en el proyecto, ya sea un gráfico para el fondo de la ventana, un icono o un cursor. En la parte superior del diseñador aparecen las distintas herramientas que es posible utilizar para, por ejemplo, dibujar líneas, borrar, insertar texto, etc.

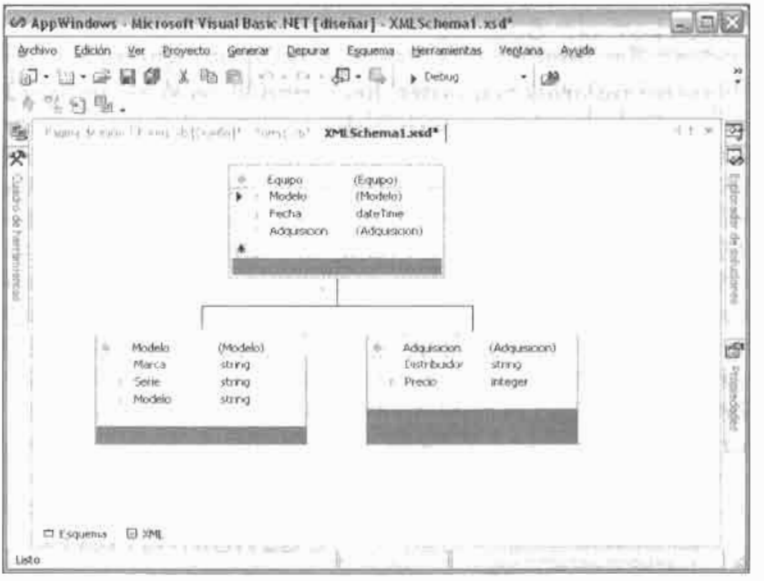


Figura 2.14. El diseñador de esquemas XML

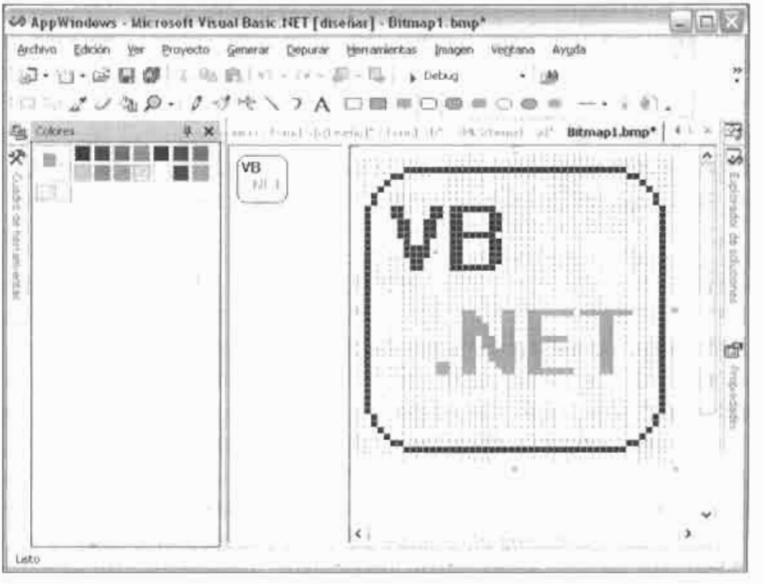


Figura 2.15. Aspecto del diseñador de imágenes de mapas de bits

Nota

En lugar de usar el diseñador de imágenes de Visual Basic .NET, puede utilizar cualquier aplicación externa, más especializada, y a continuación importar el archivo en la solución mediante las opciones del menú emergente.

Los formularios Windows, los formularios Web, las páginas HTML, los esquemas XSL, los documentos XML y, en general, si exceptuamos el caso de las imágenes, todos los elementos manipulados visualmente mediante diseñadores son la representación de un código. Ese código puede, en el caso de los formularios Windows, estar escrito en distintos lenguajes, mientras que en los documentos XML, XSL o HTML el lenguaje será, lógicamente, XML y HTML.

El editor de código es siempre el mismo, si bien su comportamiento se adapta al tipo de documento sobre el que está trabajándose. Los distintos elementos sintácticos del documento se diferencian utilizando colores y estilos. Las listas desplegables y ventanas flotantes ofrecen información sobre los miembros y parámetros de cada elemento a medida que va escribiéndose. En la figura 2.16, por ejemplo, puede ver cómo al editar un documento XML, basado en el esquema de la figura 2.15, el editor, al introducir el símbolo de apertura de marca, automáticamente nos muestra las que podemos utilizar en ese ámbito.

Figura 2.16. Edición de un documento XML

Aparte de los elementos indicados, dependiendo del tipo de documento también notará diferencia en los botones que aparecen justo encima del editor. Al trabajar con un documento XML, tenemos botones para verificar la validez del documento o mostrarlo en el explorador. Esta barra de botones es diferente cuando, por ejemplo, está editándose el código Visual Basic de un formulario Windows.

El editor de código de Visual Basic .NET se utiliza como cualquier editor de texto, utilizando las mismas teclas y combinaciones de teclas para desplazarse carácter a carácter, palabra a palabra, línea a línea, página a página, etc. Tam-

bién usaremos las mismas combinaciones para copiar, pegar, eliminar, etc. También puede utilizar el ratón y sus botones para seleccionar porciones de texto y, mediante operaciones de arrastrar y soltar, efectuar cambios.

Además de esta funcionalidad básica, el editor de Visual Basic .NET también dispone de comandos que simplifican la aplicación de formato al código, por ejemplo indentándolo adecuadamente u ocultando estructuras como los métodos o las clases para facilitar una vista esquemática del contenido. Al efectuar operaciones con el portapapeles podemos utilizar las habituales combinaciones de tecla, pero también recurrir al anillo del portapapeles que aparece en el Cuadro de herramientas.

Si bien la función principal del editor de código es, como su propio nombre indica, permitir la edición de archivos de código, mediante las opciones existentes en el menú contextual puede utilizarse este mismo editor para controlar algunos aspectos de la depuración del proyecto.

Otras herramientas

Los editores y diseñadores, conjuntamente con la ventana Propiedades y el Cuadro de herramientas, son los elementos que podríamos considerar básicos del entorno, ya que resultan indispensables para realizar nuestro trabajo: desarrollar aplicaciones. No son, sin embargo, los únicos elementos, hay otros muchos y, lógicamente, todos ellos son útiles.

Algunas de las ventanas que van a mencionarse a continuación no están visibles en principio. Tendrá que utilizar la correspondiente opción del menú Ver o del submenú Ver>Otras ventanas para hacerlas aparecer.

El Explorador de servidores

Al desarrollar una aplicación podemos necesitar componentes adicionales, no disponibles en el Cuadro de herramientas, como puedan ser conexiones con bases de datos, colas de mensajes y otros servicios que haya instalados en el sistema. En lugar de utilizar una herramienta externa, o escribir el código necesario para acceder a esos servicios, podemos servirnos del Explorador de servidores. Esta ventana aparece, inicialmente, en el margen izquierdo del entorno de Visual Basic .NET, pero ya sabe que puede recurrir al menú de opciones para hacerla aparecer.

En la ventana Explorador de servidores encontraremos, en principio, dos elementos: Conexiones de datos y Servidores. El primero de ellos actúa como raíz de todas las conexiones de datos que podamos tener definidas, mientras que el segundo es la raíz de los servidores, ordenadores, a los que tengamos acceso. La lista de conexiones de datos en principio estará vacía, mientras que en la de servidores aparecerá el ordenador local.

Utilizando las opciones del menú emergente de cada nodo podremos definir nuevas conexiones de datos, así como registrar otros servidores que pueda

haber en nuestra red para acceder a sus servicios. También puede usar con este fin dos de los botones que hay en la parte superior de la ventana, concretamente los que tienen un signo + junto al icono de una base de datos y un ordenador.

Desplegando una conexión de datos podrá acceder a los distintos elementos que haya en la base de datos: tablas, vistas, procedimientos almacenados, etc. Análogamente, abriendo la rama de un servidor tendremos acceso a sus servicios: colas de mensajes, servidores de datos y servicios de informes entre otros. En la figura 2.17 puede ver abiertas las dos ramas mostrando una conexión de datos y los servicios del sistema local. Usando la técnica de arrastrar y soltar puede insertar en el diseñador cualquiera de los elementos mostrados en el **Explorador de servidores**, creando adaptadores de datos, informes o controladores de servicios.

Figura 2.17. Ventana del Explorador de servidores

Tareas pendientes

Otro de los elementos nuevos de Visual Basic .NET, inexistente en versiones previas, es la lista de tareas. En dicha lista aparece información recogida del código de los diferentes módulos que componen el proyecto. Cada vez que se encuentra un comentario que comienza por las palabras `TODO`, `HACK` o bien `UNDONE` el comentario es tratado como una entrada de la lista de tareas, extrayéndose su localización, estado, persona responsable, prioridad, etc. También aparecen en dicha lista otros elementos, por ejemplo mensajes de error por fallos en el código.

Los elementos que aparecen en la **Lista de tareas**, cuyo aspecto por defecto puede ver en la figura 2.18, y su orden, dependen de las opciones que se hayan seleccionado en el menú **Ver>Mostrar tareas**. También es posible usar el menú emergente, como se aprecia en la citada figura, para seleccionar la lista de tareas a mostrar. Igualmente, podemos modificar el orden con tan sólo pulsar en el encabezado de una de las columnas.

Partiendo de la lista de tareas, basta un doble clic sobre cualquiera de los elementos para acceder al punto exacto del código en el que se espera la ac-

ción. De esta forma es mucho más fácil llevar un control de los trabajos pendientes que hay en el proyecto.

Figura 2.18. Aspecto de la Lista de tareas

La ventana de comandos

Visual Basic .NET cuenta con un entorno totalmente configurable, en el que cada objeto de los que ve puede ser utilizado programáticamente, desde código, como haríamos con cualquier otro objeto de una aplicación. Mediante la Ventana de comandos (véase la figura 2.19) es posible usar esos objetos para, de manera interactiva, efectuar cualquier operación que deseemos. Es posible, por ejemplo, iniciar un nuevo proyecto o abrir un archivo sin necesidad de recurrir a las opciones del menú de Visual Basic .NET.

La Ventana de comandos resulta interesante si vamos a programar macros o crear proyectos de añadidos para el entorno de Visual Basic .NET, ya que nos permiten comprobar el funcionamiento de los distintos objetos que forman parte de ese entorno de manera directa. También será útil durante la depuración de una aplicación para, por ejemplo, comprobar o modificar el valor de una variable o ejecutar en modo intérprete cualquier sentencia que pudiera interesarnos.

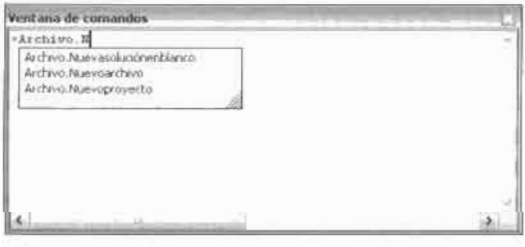


Figura 2.19. Introducimos una orden en la Ventana de comandos

La ayuda de Visual Basic .NET

Visual Studio .NET es un entorno de cierta complejidad en el que podemos utilizar múltiples lenguajes, entre ellos Visual Basic .NET, infinidad de componentes y, por supuesto, todos los servicios disponibles en la plataforma Microsoft .NET. Cada uno de estos elementos cuenta con su respectiva ayuda, recurso indispensable para poder aprovechar todas sus posibilidades. La ayuda se encuentra en los CD-ROM 2 y 3 del producto.

Nota

Recuerde que este libro se ha redactado usando una versión preliminar de Visual Studio .NET, por lo que la distribución del contenido en los CD-ROM del producto definitivo podría ser distinta.

Todas las opciones relativas a la ayuda las encontrará en el menú del mismo nombre. Desde él podrá acceder a la tabla de contenidos, buscar una cierta palabra clave, establecer filtros, etc. La ayuda se muestra, por defecto, como una página más en el área de contenido del entorno, igual que lo haría un diseñador o el editor de código.

Una novedad, relativa a la ayuda, es la existencia de la ventana Ayuda dinámica, que puede ver en la parte inferior derecha de la figura 2.20. Si mantiene abierta esta ventana verá que, a medida que introduce código, selecciona componentes o abre ventanas, aparece una lista de temas de ayuda relacionados con el contexto en el que se encuentre. Lógicamente, puede seguir utilizando la tecla **F1** para acceder a una ayuda sensible al contexto, pero la ventana Ayuda dinámica siempre aporta más información.

Con las opciones del menú Ayuda también puede acceder al servicio técnico de Microsoft, así como obtener información relativa a la versión concreta de Visual Basic .NET o Visual Studio .NET que esté utilizando, los lenguajes y herramientas instalados, etc. En la figura 2.21, por ejemplo, se puede ver que el entorno de desarrollo es la versión 7.0.9254 y que, aparte de Visual Basic .NET, también están disponibles Visual C# .NET, Visual C++ .NET, Crystal Reports para Visual Studio .NET y Application Center Test.

Adaptación del entorno

El entorno de Visual Basic .NET es personalizable y automatizable, de tal manera que podemos adaptarlo a nuestras preferencias haciendo más fácil y rápido nuestro trabajo habitual. Al principio del capítulo se indicó cómo modificar la disposición y comportamiento de las ventanas, con las que tenemos que trabajar continuamente, y poco más. En este punto nos ocuparemos con mayor detalle de la personalización de este entorno.

Figura 2.20. La ayuda de Visual Studio.NET

Figura 2.21. Información sobre el entorno de desarrollo y herramientas instaladas

Recuerde que mediante el apartado **Mi perfil** de la **Página de inicio** se puede seleccionar una configuración por defecto para la disposición de las ventanas y las combinaciones de teclado.

Partiendo de esa configuración por defecto, sin embargo, podemos efectuar los cambios que consideremos oportunos.

Nota

Recuerde que puede hacer aparecer la Página de inicio mediante la opción Ver>Explorador Web>Inicio del menú principal.

Opciones del entorno

El entorno de Visual Basic .NET no sólo se compone de ventanas que, como acabamos de ver, podemos situar donde más nos convenga y cuyo comportamiento podemos personalizar, también existen otros elementos y multitud de opciones para configurarlos. Todas esas opciones las encontrará en la ventana que se muestra en la figura 2.22, a la que puede acceder mediante la opción Herramientas>Opciones.

Figura 2.22. Ventana con opciones generales del entorno

La lista que hay a la izquierda muestra múltiples carpetas con elementos, cada uno de los cuales corresponde a un conjunto de opciones que aparecen en el espacio que hay a la derecha. Como puede ver, la lista es bastante extensa y, de acorde con ello, el número de opciones muy importante. Con ellas puede prácticamente personalizar cualquier elemento: los estilos y colores que se utilizan para realzar la sintaxis en el editor de código, las combinaciones de teclas asociadas a opciones y acciones, las palabras clave que reconocerá la lista de tareas, etc. Como se aprecia en la parte inferior de la figura 2.22, siempre tenemos a nuestra disposición el botón **Ayuda** para aprender más sobre el significado o la utilidad de las opciones que aparecen en la ventana.

En la parte superior de la página General de la carpeta Entorno, que es la que aparece abierta por defecto, encontramos unos botones que nos permiten alternar entre los dos mecanismos posibles de distribución de las ventanas del entorno: usando ventanas adosadas o ancladas, que es el modo por defecto, o bien utilizando un entorno clásico MDI. Puede seleccionar la opción Entorno MDI y reiniciar el entorno para ver la diferencia.

Configuración de botones y menús

En la parte superior del entorno de Visual Basic .NET aparecen varias paletas de botones que facilitan el acceso rápido a las opciones más habituales, así como un menú donde se encuentran prácticamente todas las opciones disponibles. Las paletas de botones cuentan con un *asa*, en el extremo izquierdo, de la cual podemos tomarlas para llevarlas a cualquier otro punto, disponiéndolas horizontal o verticalmente en los márgenes del entorno o bien dejándolas como ventanas flotantes.

La configuración de las paletas de botones no es estática, podemos tanto ocultar como mostrar paletas de botones, así como modificar los botones contenidos en ellas. Igual ocurre con los menús de Visual Basic .NET.

Selección de paletas visibles

Inicialmente, en el entorno de Visual Basic .NET tan sólo aparecen dos de las paletas de botones que hay disponibles. La primera de ellas, conocida como **Estándar**, aloja los botones que nos permiten guardar y abrir archivos y proyectos, efectuar las habituales tareas de edición y acceder a ventanas de uso intensivo como el **Cuadro de herramientas** o la ventana **Propiedades**. La segunda paleta, conocida como **Web**, cuenta con los botones típicos de exploración que, por ejemplo, a la hora de acceder a la ayuda nos permiten ir hacia atrás y adelante, modificar el tamaño de letra, etc.

Aparte de estas dos, Visual Basic .NET cuenta con al menos veinticinco paletas más. Puede acceder a ellas pulsando sobre las paletas de botones visibles con el botón secundario del ratón. Se desplegará entonces un menú como el de la figura 2.23, en el que puede marcar y desmarcar las paletas que desea hacer visibles. Como puede ver en esa figura, hay paletas de botones específicas para el trabajo con archivos HTML o XML, edición de imágenes, diseño de datos, etc.

Lo normal es que activemos tan sólo las barras de botones que necesitemos en cada momento ya que, de lo contrario, no tendríamos espacio suficiente en pantalla a menos que contemos con un monitor con mucho espacio de visualización y trabajando con una resolución relativamente alta. En la figura 2.24 se puede ver cómo queda el entorno de Visual Studio .NET tras activar todas las paletas de botones disponibles en principio.

Una alternativa, a la hora de mostrar y ocultar barras de botones, consiste en usar la opción **Personalizar** que aparece al final de la figura 2.23, abriendo la ventana de personalización de opciones de menú y botones. Esta ventana dispone de tres páginas diferentes, abriéndose por defecto la llamada **Barras de herramientas**. En ella, como se aprecia en la figura 2.25, existe una lista en la que aparecen todas las paletas de botones y podemos marcar y desmarcar para mostrarlas u ocultarlas, respectivamente.

Añadir y eliminar botones

Como acaba de decirse, en las paletas de botones tan sólo aparecen representadas las operaciones que se utilizan de manera más o menos frecuente. Esta

configuración, sin embargo, puede depender en parte del usuario de la herramienta, ya que no todos seguimos los mismos pasos para efectuar un cierto trabajo. Por ello, es posible tanto quitar botones existentes en una paleta como añadir otros nuevos.

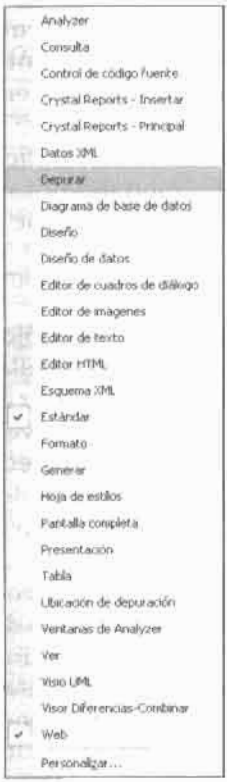


Figura 2.23. Menú para mostrar/ocultar paletas de botones

La segunda página de la ventana Personalizar, llamada Comandos, cuenta con dos listas: la de la izquierda contiene varios grupos de comandos, mientras que la de la derecha muestra los comandos existentes en el grupo seleccionado a la izquierda. Mientras tengamos abierta esta ventana, que puede ver en la figura 2.26, podemos utilizar la técnica de *arrastrar y soltar* tanto para eliminar botones de las paletas de herramientas que hay visibles como para añadirlos, tomándolos de la lista Comandos y situándolos en la posición que deseemos de la paleta.

Nota

Para eliminar un comando existente en una paleta pulse sobre él con el botón principal del ratón y arrástrelo hasta situarlo fuera de la paleta, en cualquier punto de la pantalla.

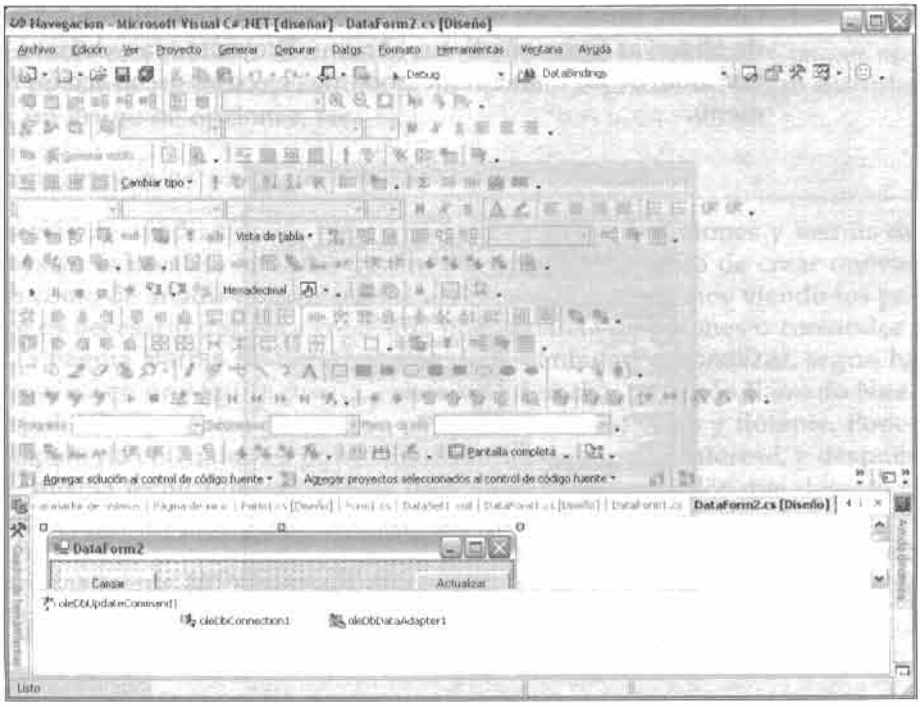


Figura 2.24. Activamos todas las paletas de botones disponibles

Figura 2.25. Otra forma de mostrar y ocultar barras de botones

Añadir y eliminar menús de opciones

Usando la misma técnica que acaba de describirse en el punto previo, arrastrando comandos a las paletas de botones y fuera de ellas, también podemos adaptar la configuración de los menús de Visual Basic. Los comandos pueden situarse en la barra de menús como nuevas opciones principales o bien como

opciones simples de un menú ya existente. En la figura 2.27, por ejemplo, puede observar cómo añadimos al menú **Proyecto**, ya existente, el comando **Guardar todo**, de tal manera que podamos ejecutar dicha acción también desde este menú.

Figura 2.26. Lista de los comandos disponibles para alojar en paletas de botones

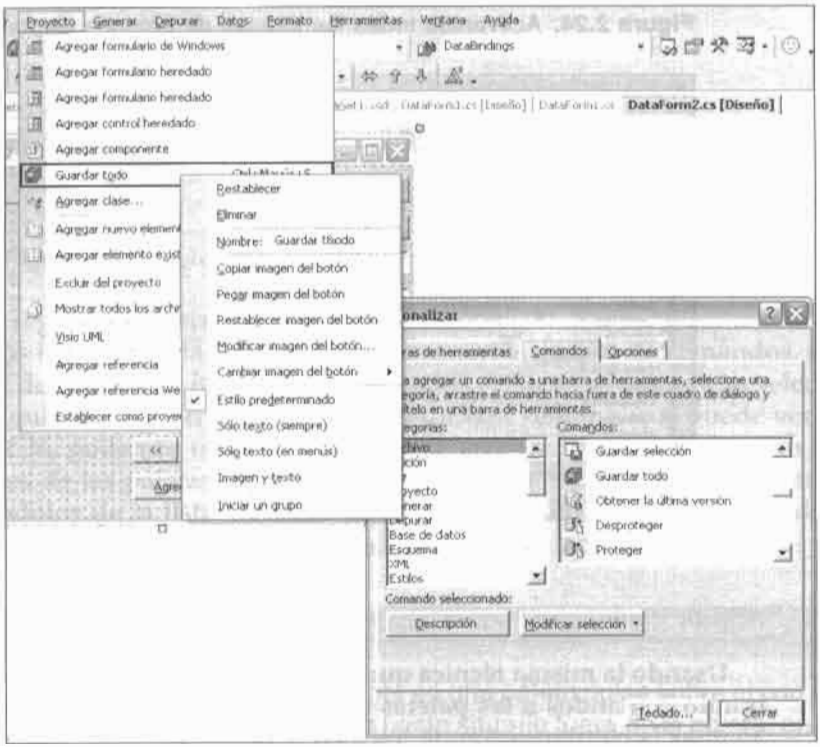


Figura 2.27. Añadimos opciones a un menú existente

El menú emergente que aparece sobre la opción recién añadida lo hemos hecho aparecer con el botón secundario del ratón, como es habitual. Con sus opciones podemos modificar el título de la opción, asignarle un icono distinto, iniciar un grupo de opciones, etc.

Creación de nuevas paletas y menús

Además de añadir nuevos elementos a las paletas de botones y menús de opciones ya existentes, también tenemos la posibilidad tanto de crear nuevas paletas como de añadir nuevos menús al entorno. Comencemos viendo los pasos que es necesario dar para crear una nueva paleta de botones o comandos.

En la página Barras de herramientas de la ventana Personalizar, según ha podido verse en una figura previa, existe, a la derecha, un botón llamado **Nueva**. Al pulsarlo aparecerá una nueva paleta de botones, vacía y flotante. Podemos disponerla en cualquier punto del entorno, donde nos interese, y después usar la misma técnica descrita antes para añadir los comandos que deseemos tener en esa nueva paleta.

La figura 2.28 muestra una paleta de herramientas, a la que hemos llamado muy originalmente MiPaleta, que dispone de cinco botones con tareas que, se supone, realizamos habitualmente.

Figura 2.28. Una nueva paleta de botones

Si lo que deseamos es crear un nuevo menú, tendremos que seleccionar el último elemento de la lista Categorías en la página Comandos de la misma ventana. A la derecha verá aparecer un único comando: Nuevo menú. Arrástrelo hasta situarlo en la barra de menús, como se ha hecho en la figura 2.29, o, si lo prefiere, en el interior de una barra de botones, creando así un menú desplegable. Una vez creado el menú, pulse el botón secundario del ratón sobre él para establecer el nombre y modificar cualquier otra propiedad.

Añadido el menú, el proceso para insertar en él opciones u otros menús es idéntico al descrito en el punto previo. En la figura 2.30 puede ver el menú tras haber añadido cinco opciones.

Opciones diversas

La ventana Personalizar cuenta, como se decía anteriormente, con tres páginas distintas. En la tercera, titulada Opciones, encontrará distintas opciones que afectan a las barras de botones y menús de opciones estableciendo, por ejemplo, el tamaño de los iconos, los efectos de animación al desplegar un menú, la aparición de las combinaciones de tecla al situar el puntero del ratón sobre un botón, etc.

Quizá lo más interesante de esta ventana sea el botón **Teclado** el cual aparece en la parte inferior. Al pulsarlo verá aparecer una ventana como la de la figura 2.31. En realidad es la ventana de opciones del entorno que aparece con el

elemento Entorno>Teclado ya seleccionado. En la lista que ocupa la parte central aparecen todos los comandos disponibles en el entorno, mientras que justo debajo se indican las combinaciones de tecla para ejecutarlos sin necesidad de recurrir a botones ni opciones. Lógicamente, podemos tanto modificar las asignaciones existentes como añadir atajos de teclado a opciones que no contasen con ellos.

Figura 2.29. Añadimos un nuevo menú al entorno

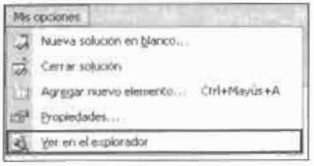


Figura 2.30. Nuevo menú de opciones

Configuración de herramientas externas

Aunque Visual Basic .NET cuenta, como ha podido ver y verá en los distintos capítulos de este libro, con un entorno que tiene infinidad de posibilidades, está claro que no puede integrar absolutamente todos los recursos que un pro-

gramador pudiera necesitar para efectuar su trabajo, desde el primero hasta el último. Es por eso que desde Visual Basic .NET resulta posible acceder a herramientas externas, para lo cual tan sólo hay que definir la configuración apropiada para ejecutarlas.

Figura 2.31. Configuración de los atajos de teclado

Si abre el menú **Herramientas** encontrará en él opciones para ejecutar algunas herramientas externas, como comprobador de controles ActiveX, el generador de GUID o bien el Visor de objetos OLE y COM. Éstas son herramientas externas ya preconfiguradas. Para añadir cualquier herramienta adicional utilice la opción **Herramientas externas** de ese mismo menú, abriendo la ventana mostrada en la figura 2.32. En ella aparece una lista con todas las aplicaciones configuradas en este momento. Al seleccionar cualquiera de ellas, en la parte inferior aparecen el título, comando de ejecución y argumentos, entre otros datos. Los botones que aparecen a la derecha permiten añadir y eliminar opciones del menú, así como establecer el orden.

Figura 2.32. Ventana Herramientas externas

Pulsando el botón **Añadir** aparecerá un nuevo elemento en la lista. Usando los controles que hay en la parte inferior configuramos el título y los parámetros. Éstos pueden seleccionarse de una lista desplegable, como se aprecia en la figura 2.33. En ese caso se está configurando una opción, llamada Bloc de notas, que al ejecutarse abrirá en dicha aplicación el módulo de código fuente sobre el que estemos trabajando. De manera análoga podríamos configurar cualquier otra herramienta.

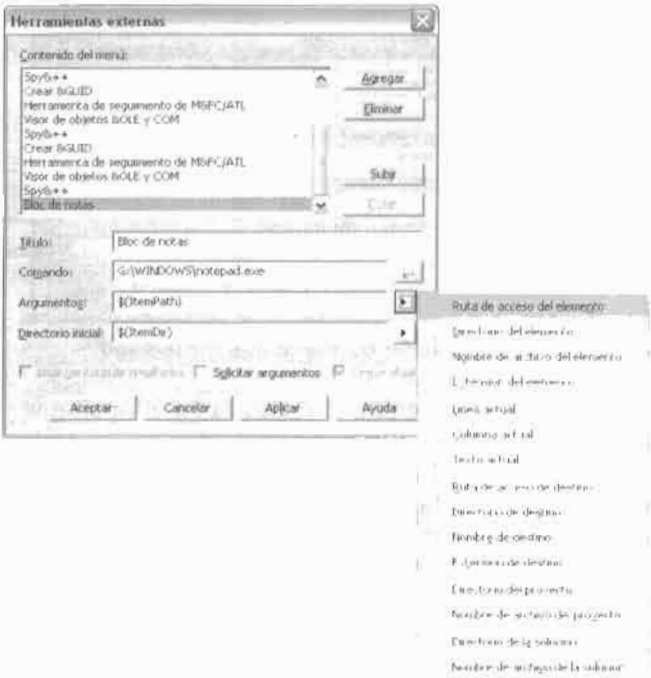


Figura 2.33. Añadimos una nueva herramienta externa

Automatización de tareas

En el trabajo diario es corriente que existan tareas más o menos repetitivas, no complejas pero en las que se emplea un cierto tiempo precisamente por la frecuencia con la que hay que realizarlas. Visual Basic .NET cuenta con diversos mecanismos para automatizar esas tareas, desde la programación de nuevos asistentes hasta el registro y reproducción de macros. Este último es, seguramente, el método más simple, pero no por ello con menos posibilidades. Es, además, un recurso nuevo, inexistente en versiones previas de Visual Basic.

Las macros pueden crearse de dos maneras distintas: registrando las acciones efectuadas con el teclado y el ratón o bien codificándolas manualmente. Una vez existen, pueden ejecutarse desde una opción de menú, con una combinación de teclado o pulsando un botón.

Registro de una nueva macro

Comencemos por el principio, registrando la que sería nuestra primera macro en Visual Basic .NET. Con este fin seleccionaríamos la opción Herramientas>Macros>Grabar TemporaryMacro. También podemos pulsar la combinación de teclas **Mayús-Control-R** para conseguir exactamente la misma acción.

Al iniciar el registro de una nueva macro veremos aparecer como ventana flotante la paleta de botones mostrada en la figura 2.34. Con el primer botón detendremos momentáneamente el registro, con el segundo lo finalizaremos y con el tercero cancelaremos la grabación.

Figura 2.34. Paleta de botones para controlar el registro de una macro

Durante el registro de la macro podemos utilizar las opciones del menú, los botones disponibles en las paletas, podemos realizar las tareas habituales de diseño en los formularios o en el código fuente. Cualquier acción será registrada y traducida a órdenes que, finalmente, serán las que compongan la macro. Puede, por ejemplo, introducir un texto cualquiera en el editor de código y copiarlo varias veces.

Ejecución de una macro

Al finalizar el registro de la macro, pulsando el segundo de los botones existentes en la paleta, ésta desaparecerá y la macro quedará almacenada en un espacio temporal. Para ejecutarla tan sólo tenemos que pulsar la combinación **Mayús-Control-P**, o bien usar la opción Herramientas>Macros>Ejecutar TemporaryMacro. Al hacerlo, verá cómo se reproduce la inserción del texto que introdujo durante el registro.

En este momento la macro está, como se ha dicho, almacenada de forma temporal. Esto significa que el registro de una nueva macro eliminaría a ésta que tenemos. Si hemos creado una macro útil, que puede servirnos en otras ocasiones e incluso otros proyectos, lo normal es que la guardemos. Para ello usaremos la opción Herramientas>Macros>Guardar TemporaryMacro. Aparecerá el Explorador de macros (véase la figura 2.35) invitándonos a asignar un nombre a esa macro temporal. De esta forma podemos guardar tantas macros como necesitemos.

Nota

Teniendo el Explorador de macros abierto, podemos ejecutar cualquier macro simplemente haciendo doble clic sobre su nombre, sin necesidad de acceder a opción de menú alguna.

Figura 2.35. Asignamos un nombre a la macro recién registrada

Código de la macro

Utilizando las opciones del menú emergente asociado a cada elemento del Explorador de macros podemos acceder al código fuente de cada una de ellas. Por ejemplo, pulsando el botón secundario del ratón sobre la macro que acabamos de registrar y seleccionando la opción Editar, nos encontraremos con una ventana como la de la figura 2.36. Es el editor de macros de Visual Basic .NET, un punto desde el cual podemos tanto explorar el código de las macros existentes como modificarlo o, incluso, crear nuevas macros directamente, introduciendo el código Visual Basic apropiado.

Para codificar una macro se utiliza el modelo de objetos de Visual Studio .NET, un modelo en el que cada elemento del entorno está representado por un objeto, un modelo en el que cada acción que realizamos mediante el teclado o ratón tiene una correspondencia con un método de un objeto. En la figura 2.36 puede ver cómo se usa el objeto `ActiveDocument` para acceder a la selección de texto del documento actual, usando métodos como `Copy()` y `Paste()` para copiarlo y pegarlo. Lógicamente, para sacar el mejor provecho de las macros es indispensable conocer a fondo este modelo de objetos, un tema que, por su extensión y complejidad, queda fuera del ámbito de este libro.

Puntos clave

- Los proyectos Visual Basic .NET se agrupan en *soluciones*. Un proyecto puede contener formularios, módulos de código, archivos de recursos y otros elementos adicionales.
- Los módulos correspondientes a elementos visuales, como formularios o imágenes, se editan usando los diseñadores existentes en el entorno, con la ayuda de ventanas como Propiedades y Cuadro de herramientas.
- El código asociado a esos elementos, así como cualquier otro código del proyecto, se manipula en un editor de texto con realce sintáctico y tecnología *Intellisense* de ayuda a la edición.

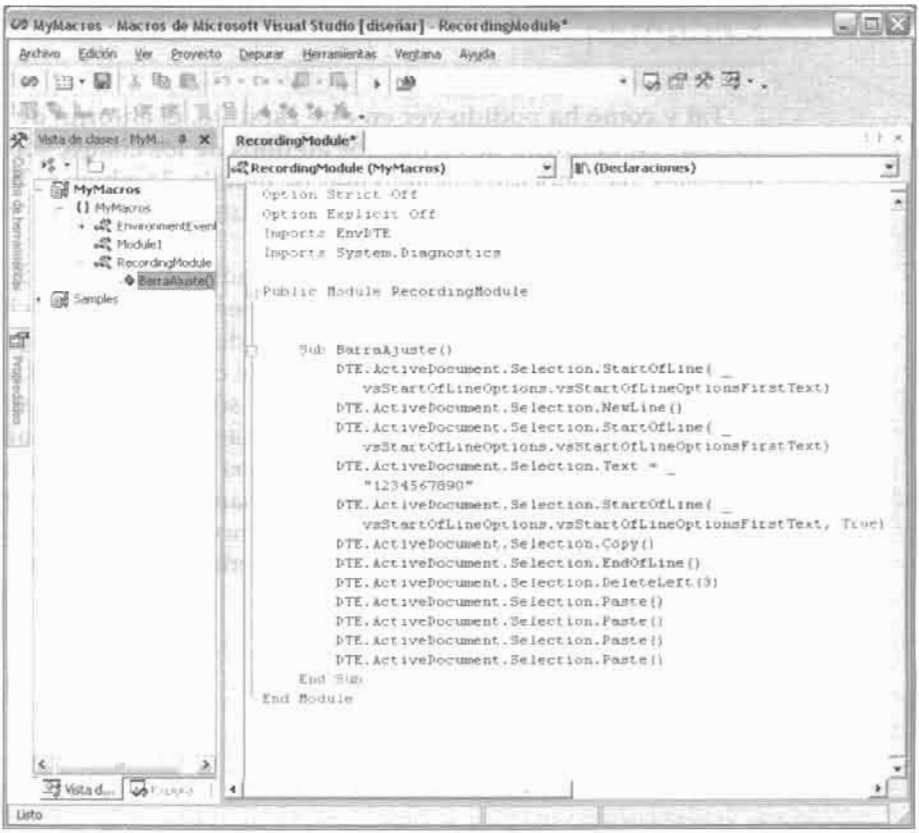


Figura 2.36. Editor de macros de Visual Studio.NET

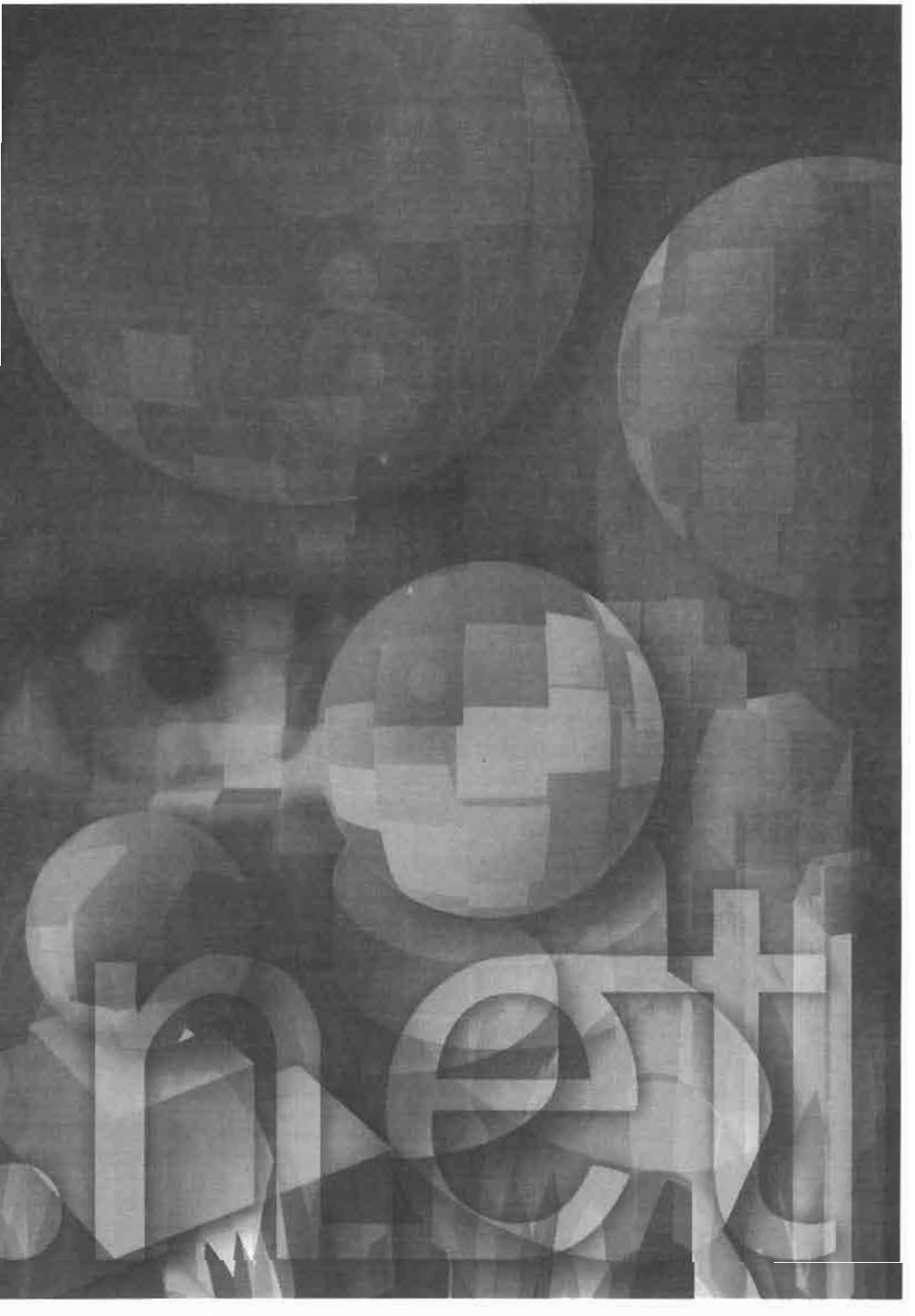
- La base del desarrollo con Visual Basic .NET sigue siendo el uso de componentes y controles. Éstos se insertan en contenedores, como por ejemplo pueden ser los formularios, y se personalizan mediante propiedades y eventos.
- El entorno de desarrollo cuenta con un modelo de objetos que facilita su control mediante comandos. Ese mismo modelo puede usarse para automatizar tareas o crear nuevos asistentes.
- Las ventanas existentes en el entorno pueden anclarse unas a otras, apilarse en múltiples páginas o quedar como flotantes, todo ello mediante operaciones de arrastrar y soltar.
- Otros elementos del entorno, como las paletas de botones y los menús de opciones, pueden adaptarse a nuestras preferencias. Recuerde utilizar el botón secundario del ratón sobre cualquier elemento para acceder a sus opciones.
- La automatización de tareas resulta posible y sencilla gracias a la nueva posibilidad de registrar y reproducir macros.

Resumen

Tal y como ha podido ver en este capítulo, el entorno de Visual Basic .NET cuenta con multitud de elementos algunos de los cuales, sobre todo los fundamentales, ha conocido con algo más de detalle. También se ha tratado la personalización de ese entorno, así como la automatización de tareas mediante el uso de macros.

Los elementos descritos con más profundidad, como los diseñadores o el editor, son seguramente los más utilizados, pero hay otros muchos que ni siquiera se han mencionado, como las ventanas de depuración, el esquema de documento, etc. Algunos de ellos tendrá ocasión de conocerlos en posteriores capítulos.

En el campo de la automatización tan sólo se ha abordado el registro y reproducción de una macro simple, una macro que hemos almacenado con un nombre para poder usarla siempre que nos interese. Conociendo el modelo de objetos de Visual Studio .NET tenemos también otras posibilidades, como la creación de asistentes o añadidos al entorno. En la ventana de creación de nuevos proyectos encontrará opciones que generan un esqueleto para este tipo de elementos.



3
La plataforma Microsoft .NET

El desarrollo de aplicaciones en general, y en particular en el entorno del sistema operativo Windows, ha cambiado mucho en la última década, desde la aparición de la primera versión de Visual Basic, y, especialmente, desde la difusión a escala global de Internet así como la mayor presencia de dispositivos móviles.

Hace diez años las aplicaciones se creaban usando un mismo lenguaje para todas las tareas y para un sistema operativo concreto, escenario que ha ido haciéndose más y más complejo en los últimos años.

Las primeras aplicaciones para Windows se desarrollaban usando el lenguaje C y llamadas directas a las funciones del API de Windows. Esta forma de exponer los servicios propios del sistema, a través de cientos o miles de funciones independientes, era general a todos los sistemas operativos. El trabajo era realmente complejo, siendo necesario el trabajo intensivo con punteros y manejadores (*handles*), para facilitar parámetros de la aplicación al sistema y viceversa, con el consiguiente peligro de incurrir en fallos más o menos típicos: la no liberación de bloques de memoria que quedan perdidos, reutilización de un puntero con direcciones de datos ya liberados, uso del manejador inválido al llamar a una función, etc.

Para simplificar este esquema, y ayudar a los programadores haciendo más fácil su trabajo, se buscaron soluciones siguiendo caminos paralelos, e incluso a veces divergentes, dentro de la propia Microsoft.

Así, por una parte nació Visual Basic para facilitar el desarrollo de aplicaciones simples con intertaz de usuario, ocultando todos los detalles de comunicación con el sistema. Tan sólo había que usar unos controles, conocidos como

VBX, para diseñar las ventanas y utilizar un mecanismo de propiedades y eventos para controlar el flujo de la aplicación.

En contraposición, a los programadores que utilizaban C se les ofreció como solución el lenguaje C++, orientado a objetos pero con la misma complejidad y debilidad que C, y una biblioteca de clases, llamada MFC (*Microsoft Foundation Classes*), que hasta cierto punto encapsulaba en objetos la funcionalidad usada más habitualmente del API de Windows.

Paralelamente a Visual Basic y las MFC surgió COM (*Component Object Model*), un modelo de objetos que tenía por objetivo hacer posible la reutilización de componentes binarios sin importar las diferencias entre lenguajes. COM, no obstante, siempre ha sido muy complejo y su éxito se debe, en parte, al éxito de Visual Basic 5 y 6, con los que resultaba muy fácil crear este tipo de componentes, y el ofrecimiento de una biblioteca de plantillas, la ATL (*Active Template Library*), que, conjuntamente con una serie de asistentes, allanaba su desarrollo en C++.

Nuevas necesidades

Todo el escenario que acaba de exponerse se ciñe al desarrollo de aplicaciones para un sistema operativo: Windows. Este escenario adquiere mayor complejidad debido a dos factores fundamentales: la necesidad de aprovechar las posibilidades que ofrece la red Internet y la cada vez mayor presencia, ¿necesidad?, de llegar con la información a todos lados mediante dispositivos móviles tales como teléfonos celulares, asistentes digitales personales y dispositivos parecidos.

Para cubrir estas necesidades, relativamente recientes, también se han seguido varios caminos paralelos. Con el fin de crear aplicaciones para Internet puede utilizar ASP (*Active Server Page*), conjuntamente con un lenguaje de guiones como JScript o VBScript, o bien ISAPI (*Internet Information Server API*), usando C++ para crear módulos que sólo funcionan con IIS. Con Visual Basic 6, gracias a la integración de COM tanto en IIS como en el lenguaje, se facilitaba el desarrollo de este tipo de proyectos.

En cuanto a las aplicaciones móviles, prácticamente se ha contado con los mismos entornos de desarrollo existentes para Windows, como Visual C++ y Visual Basic, pero modificados para ajustarse a las limitaciones de un sistema operativo embebido como es el usado en los Pocket PC.

Con el fin de agrupar bajo un mismo nombre todos los servicios necesarios para este tipo de aplicaciones, servicios como ASP, ADO o los servicios de componentes COM+, nace la denominación Windows DNA (*Distributed Internet Architecture*).

En resumen, podríamos decir que a medida que han ido surgiendo las necesidades se han hecho añadidos tanto al sistema operativo como a las herramientas de desarrollo, añadidos heterogéneos que en unas ocasiones vienen en forma de más funciones tipo API, en otras como componentes COM que se integran en el sistema y en otras como asistentes para los entornos de desarrollo.

.NET, ¿la solución definitiva?

El resultado de esta década de evolución en el desarrollo de aplicaciones, en general y particularmente en el entorno de Windows, ha tenido como resultado la existencia actualmente de multitud de servicios duplicados, específicos para un determinado lenguaje o herramienta, y pocas posibilidades reales de reutilización del código, al tiempo que el propio sistema ha ido ganando en complejidad de manera vertiginosa.

La solución, quizá definitiva, que propone Microsoft para solventar esta situación es la plataforma .NET. Ésta consta de un conjunto de servicios universales, idénticos para todos los lenguajes, al tiempo que mantiene la integración con los desarrollos existentes y hace posible una interoperabilidad entre lenguajes desconocida hasta el momento. Aunque actualmente la plataforma .NET sólo está disponible para Windows 98, Me, NT, 2000 y XP pronto aparecerá una versión compacta para dispositivos móviles, al tiempo que no se descarta la aparición en otros sistemas operativos, como Linux.

Para desarrollar aplicaciones .NET puede usarse prácticamente cualquier lenguaje de programación, desde C++ y Visual Basic hasta el nuevo Visual C#, pasando por COBOL, Perl, Pascal, Java o Python. No tenemos, por tanto, que utilizar obligatoriamente un determinado lenguaje. Los servicios de la plataforma, tanto la biblioteca de componentes como los servicios básicos, son expuestos de forma idéntica a todos los lenguajes, de tal manera que todos ellos tienen las mismas posibilidades. Podemos, por tanto, crear un componente de servidor con múltiples hilos de ejecución utilizando Visual Basic .NETo una interfaz Web sirviéndonos de Visual C++ .NET.

La plataforma .NET no sólo hace posible el desarrollo de componentes y aplicaciones para Windows, que sigue siendo uno de los primeros objetivos de la mayoría de los programadores, sino también la creación de aplicaciones y servicios Web, aplicaciones de consola, bibliotecas de clases genéricas, aplicaciones para dispositivos móviles, etc. Es, por tanto, un recurso adaptado a las necesidades actuales, frente a las soluciones antiguas y reiteradamente actualizadas o *parcheadas* a lo largo de los años.

Desde las aplicaciones .NET puede seguir utilizándose el API de Windows y los componentes COM. De manera análoga, los componentes .NET pueden ser utilizados por aplicaciones clásicas también a través de COM. No se rompe, por tanto, con los proyectos ya existentes forzando una actualización global. Usando los servicios de interoperabilidad cualquier empresa puede ir desarrollando sus nuevas aplicaciones en .NET integrándolas con las que ya tiene.

Para terminar este punto, destacar que el desarrollo general de aplicaciones, sin importar su naturaleza, resulta mucho más sencillo en la plataforma .NET de lo que era anteriormente utilizando el API de Windows, MFC o Visual Basic. Esto es así porque todos los servicios se exponen en forma de componentes, dado que .NET es una plataforma diseñada con una orientación a componentes en mente. Podemos olvidarnos de los manejadores, los punteros y la gestión de bloques de memoria, centrándonos directamente en la funcionalidad de nuestra aplicación.

Esquema de desarrollo y ejecución

Vamos a comenzar trazando un esquema general del desarrollo y ejecución de las aplicaciones en la plataforma Microsoft .NET para, en los puntos siguientes, ir profundizando en los detalles de algunos puntos más concretos.

Las herramientas de desarrollo para la plataforma .NET, en nuestro caso Visual Basic .NET, generan a partir de los diferentes módulos que conforman un proyecto los respectivos módulos de código objeto. Éste se aloja, como es tradicional, en archivos con extensiones `EXE` y `DLL` aunque, en realidad, la estructura interna de dichos archivos no es la clásica y el código que contienen no es directamente ejecutable, de ahí que no puedan usarse sobre Windows si no se tiene instalada la plataforma Microsoft .NET.

Cada uno de los módulos objeto contiene código independiente del tipo de procesador y sistema operativo. Este código, llamado código en MSIL (*Microsoft Intermediate Language*), no puede ejecutarse sin más, sino que tiene que ser previamente compilado para el sistema y procesador concretos donde va a utilizarse la aplicación. Al compilador que efectúa esa compilación *al vuelo*, cuando es necesario, se le conoce como compilador JIT (*Just in time*).

En la plataforma .NET la unidad mínima que puede alojarse en memoria no es la librería de enlace dinámico ni el ejecutable, sino que es el *assembly*, término que podemos traducir por *montaje* o *ensamblado*. Un *assembly* puede ser un solo archivo o bien estar compuesto de múltiples archivos con dependencias entre ellos.

Una misma aplicación puede constar de múltiples ensamblados, de igual forma que un programa Windows clásico puede componerse de varias librerías de enlace dinámico. Todos los ensamblados de una aplicación se ejecutan en el contexto de un mismo *application domain* o dominio de aplicación. El concepto de dominio de aplicación es una evolución del concepto de proceso tal y como se conoce actualmente en Windows, es un contexto en el que se comparten ciertos recursos: espacio de direccionamiento, modelo de hilos, etc.

Para generar el código MSIL, los compiladores hacen uso del CTS (*Common Type System*), un sistema en el que se definen los tipos de datos y operaciones posibles recogiendo la sintaxis de muchos lenguajes. Si se desea que el código pueda cooperar con otros desarrollos, sin importar el lenguaje en que se escriban, entonces habrá que ajustarse a la CLS (*Common Language Specification*), una especificación en la que se define un subconjunto del CTS que es común a todos los lenguajes.

El código MSIL contenido en los ensamblados es alojado en memoria, a demanda del usuario o de otras aplicaciones, por el CLR (*Common Language Runtime*). El CLR, o entorno común de ejecución, es una de las piedras angulares de la plataforma .NET. Se encarga no sólo de abrir los ensamblados, preparar el dominio de aplicación necesario y alojar el código en memoria, sino que, además, es el responsable de supervisar ese código encargándose, por ejemplo, de recolectar los objetos en desuso liberando la correspondiente memoria, un proceso que se conoce habitualmente como GC (*Garbage Collection*) o recolección de basura.

Para convertir el código MSIL en código ejecutable, el CLR utiliza un compilador JIT. Éste optimizará el código para el sistema operativo y procesador en que va a ejecutarse, de tal manera que la aplicación siempre aprovechará de la mejor forma los recursos del entorno donde vaya a ejecutarse. Antes de compilar, sin embargo, el CLR verifica el código para comprobar que es seguro y puede ser apropiadamente supervisado en ejecución.

La figura 3.1 resume esquemáticamente algunos de los pasos que se han mencionado. Éstos se han separado en dos grandes bloques: desarrollo y ejecución. Para el desarrollo dispondremos de Visual Basic .NET, utilizando sus diseñadores y editores para generar el código fuente que, finalmente, compilaremos obteniendo los correspondientes ensamblados. Cuando éstos sean requeridos, para ejecutar la aplicación, el código MSIL que contienen será verificado y compilado a código nativo, ejecutándose en la plataforma .NET bajo supervisión del CLR.

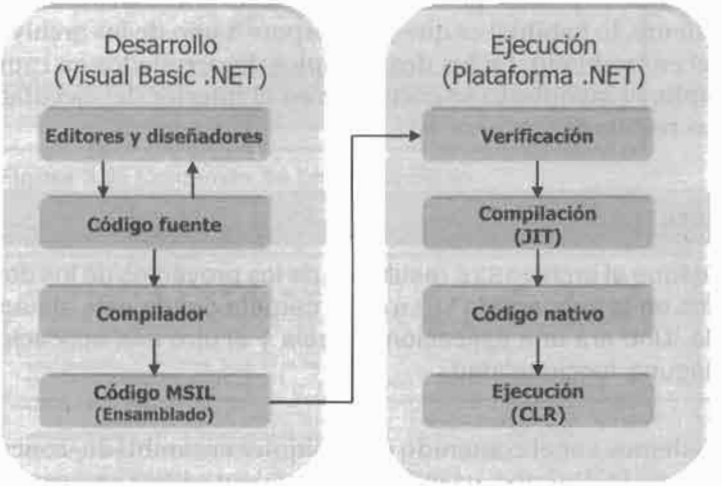


Figura 3.1. Esquema general de trabajo de una aplicación .NET

Resta decir que las aplicaciones .NET harán uso de los servicios de la plataforma Microsoft .NET, lo que se conoce también como *Microsoft .NET Framework*. Estos servicios proveen de todo lo necesario para aplicaciones clásicas, aplicaciones Web, servicios web, acceso a datos, etc.

Composición de una aplicación .NET

Una aplicación .NET consta de uno o más ensamblados, cada uno de los cuales estará formado de uno o más archivos conteniendo recursos y código MSIL. La aplicación, por tanto, puede ser tan sencilla como para estar compuesta de un simple archivo `EXE` o `DLL`, o tan compleja como para constar de múltiples archivos con código y otros elementos.

Las aplicaciones .NET se ejecutan en el contexto de un dominio de aplicación, un ámbito virtual mediante el que se aísla a esa aplicación de otras que pudieran estar ejecutándose en el sistema. Un mismo proceso de Windows puede contener múltiples dominios de aplicación sin que, por ello, existan conflictos entre ellos.

Ensamblados y manifiestos

Si un ensamblado puede estar compuesto de múltiples módulos, cabe preguntarse cómo sabrá el sistema, a la hora de utilizar dicho ensamblado, qué módulos son los que forman parte de él. La respuesta la encontramos en una porción de información, conocida como manifiesto, que relaciona a todos los elementos del ensamblado y, además, señala las dependencias existentes respecto a otros ensamblados, por ejemplo con servicios del sistema.

Aunque es posible generar el manifiesto almacenándolo en un archivo independiente, lo habitual es que se incorpore a uno de los archivos que forman parte del ensamblado. En los dos ejemplos desarrollados en capítulos previos, por ejemplo, el manifiesto se encuentra en el interior del ejecutable obtenido como único resultado.

Nota

Busque el archivo `EXE` resultante de los proyectos de los dos capítulos previos en la subcarpeta `\bin` de la carpeta donde esté almacenado el ejemplo. Uno era una aplicación consola y el otro una aplicación Windows sin ninguna funcionalidad.

Podemos ver el contenido de cualquier ensamblado, concretamente el manifiesto que lo describe, usando la herramienta `ildasm` mencionada en el primer capítulo. La encontrará normalmente en la carpeta `\Archivos de programa\Microsoft.NET\FrameworkSDK\Bin` de la unidad donde haya instalado la plataforma Microsoft .NET. Invoque esa utilidad pasando como parámetro el camino y nombre del ejecutable `AppWindowsForm.exe` creada como ejemplo en el segundo capítulo. Verá aparecer una ventana como la de la figura 3.2. En ella se enumeran los elementos que existen: un manifiesto y un ámbito con nombre que contiene una clase.

Haciendo doble clic sobre el elemento MANIFEST se abrirá una nueva ventana mostrando el manifiesto del ensamblado. Su contenido, como puede verse en la figura 3.3, seguramente no le dirá mucho. En las primeras líneas se hace referencia a varios ensamblados externos, concretamente los necesarios para usar formularios Windows y `mscorlib`, que contiene los servicios básicos del entorno común. Si abre este mismo proyecto en el entorno de Visual Studio .NET y abre el Explorador de soluciones, verá en la lista de archivos uno llamado `AssemblyInfo.vb`. Este módulo contiene la información del ensamblado que se utiliza para crear el manifiesto.

Figura 3.2. Contenido de un ensamblado

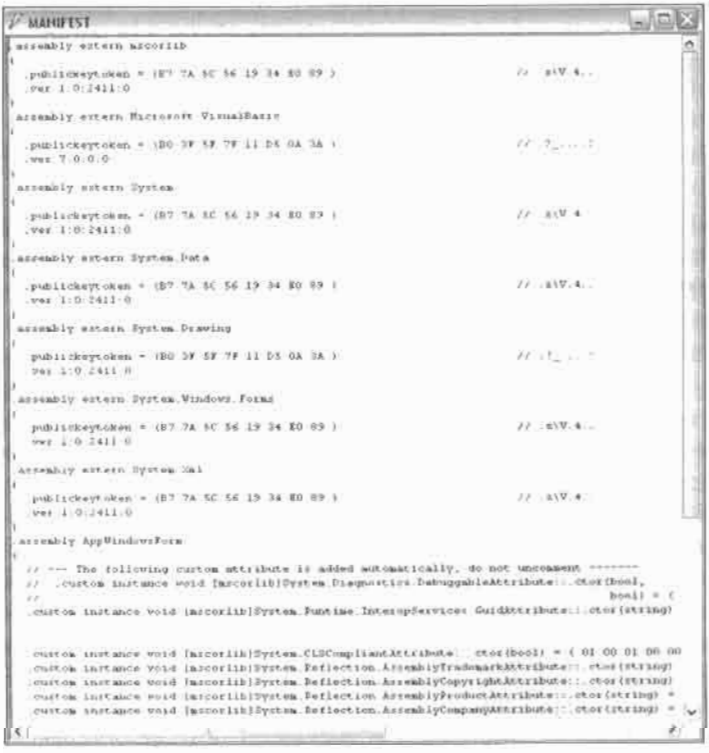


Figura 3.3. Contenido del manifiesto

Ensamblados y dominios de aplicación

Un ensamblado, como ya se ha dicho, contiene el código y los recursos de una aplicación, ya sea en un solo archivo o en varios. Hasta cierto punto, podríamos establecer una analogía con cualquier aplicación Windows tradicional, compuesta de un ejecutable principal y, opcionalmente, código adicional distribuido en módulos `DLL`.

Cuando una aplicación Windows tradicional se pone en marcha, el sistema crea un nuevo proceso y un primer hilo de ejecución para ejecutarla. El proceso define el ámbito de los recursos disponibles para la aplicación, por ejemplo el espacio de direcciones privado, evitando conflictos con otras aplicaciones que, a su vez, contarían con otro proceso.

Al iniciar una aplicación .NET el CLR, o entorno común de ejecución, crea un nuevo dominio de aplicación o `AppDomain`. Cada aplicación alojará en su `AppDomain` los ensamblados que necesite. A diferencia de lo que ocurre con las aplicaciones tradicionales, el aislamiento de las aplicaciones no precisa el uso de un proceso independiente. De hecho, es posible que el entorno de ejecución utilice un mismo proceso para varios `AppDomain`. No por ello, sin embargo, habrá conflictos ya que cada `AppDomain` mantiene totalmente aislada a su aplicación de las demás.

La figura 3.4 es una representación esquemática de lo que podrían ser dos aplicaciones .NET. Ambas se ejecutan en el mismo proceso, pero cada una en su propio dominio. La primera aplicación, arriba a la izquierda, es nuestro ejemplo `AppWindowsForm` que, como ya sabemos está compuesta de un solo ensamblado que, a su vez, consta de tan sólo un módulo ejecutable. La segunda aplicación, hipotéticamente consta de dos ensamblados. El primero está compuesto de un ejecutable y algún módulo `DLL`, mientras que el segundo es una `DLL` adicional.

Un mismo ensamblado puede, en la práctica, ser usado por distintas aplicaciones, de tal forma que se alojaría en el dominio de cada una de ellas. El entorno de ejecución se encargará de compartir el código del ensamblado entre los dominios, asegurando que en cada uno de ellos el ensamblado aparezca como privado.

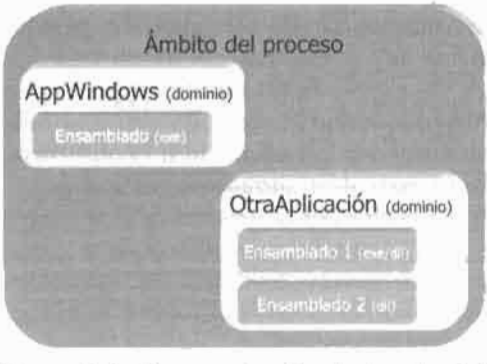


Figura 3.4. Representación de dos dominios de aplicación y sus ensamblados

Ensamblados privados y compartidos

En el punto anterior se indica la posibilidad de que un mismo ensamblado sea usado por más de una aplicación. En la práctica, esto sólo sería posible si ese ensamblado fuese de tipo *shared* o compartido. Por defecto, los ensamblados que se generan al compilar un proyecto son privados, ensamblados que sólo serán usados por la aplicación que está desarrollándose.

A diferencia de lo que ocurre con los componentes y servicios ofrecidos a través de COM, cuyos módulos deben contar con distintas entradas en el registro para que puedan ser encontradas por las aplicaciones, los ensamblados privados no precisan configuración alguna. Basta con distribuirlos conjuntamente, todos los ensamblados de la aplicación, copiándolos en el sistema de destino para que funcionen correctamente. La instalación, por tanto, es tan simple como arrastrar y soltar los archivos desde un soporte, por ejemplo un CD-ROM, al disco de destino. La desinstalación, consecuentemente, es igualmente simple, bastando con eliminar los archivos. No hay entradas huérfanas en el registro de Windows ni nada parecido.

Gracias a este comportamiento, no tendremos nunca más problemas con las versiones de las `DLL` instaladas en el sistema. Cada aplicación se distribuye con los ensamblados en la versión apropiada, sin entrar en conflicto con otras versiones que pudiesen estar usando otras aplicaciones. Adiós al conocido como *DLL hell*.

Si todos los ensamblados fuesen privados, no obstante, nos encontraríamos en una situación compleja al tener que distribuir también, junto a nuestra aplicación, todos los que contienen los servicios del sistema y el *Microsoft .NET Framework*. Para evitarlo existen los ensamblados compartidos, ensamblados que pueden ser utilizados por múltiples aplicaciones.

Los ensamblados compartidos no se distribuyen con las aplicaciones, sino que se instalan en un lugar común del sistema para que todas ellas puedan encontrarlos. Este lugar común es conocido como GAC (*Global Assembly Cache*), siendo como un depósito global de ensamblados. La localización de este depósito dependerá del sistema operativo donde esté instalada la plataforma. En Windows XP, por ejemplo, la carpeta `Windows\assembly` es la que contiene el GAC (véase figura 3.5).

Al disponer de un depósito global para los ensamblados compartidos podría pensarse en los mismos problemas que se tienen habitualmente en Windows con las versiones de las `DLL`. Afortunadamente, ese problema, el de las distintas versiones del mismo componente, no existe en la plataforma .NET. Para que un ensamblado pueda ser compartido es obligatorio que su nombre sea único, fin para el cual se le asocia una clave pública de tipo RSA y también se distingue entre versiones del mismo elemento. En la figura 3.5 puede ver que una de las columnas existentes es, precisamente, Versión. Podríamos tener ensamblados con el mismo nombre pero de distinta versión sin ningún problema, cada aplicación utilizaría la versión en la que esté interesada.

Disponemos de diversas herramientas mediante las cuales generar esas claves públicas e instalar los ensamblados compartidos en el GAC. Desde Visual

Basic .NET, como se aprecia en la figura 3.6, basta con pulsar un botón para generar la clave, introducirla en un archivo y añadir éste al proyecto actual.

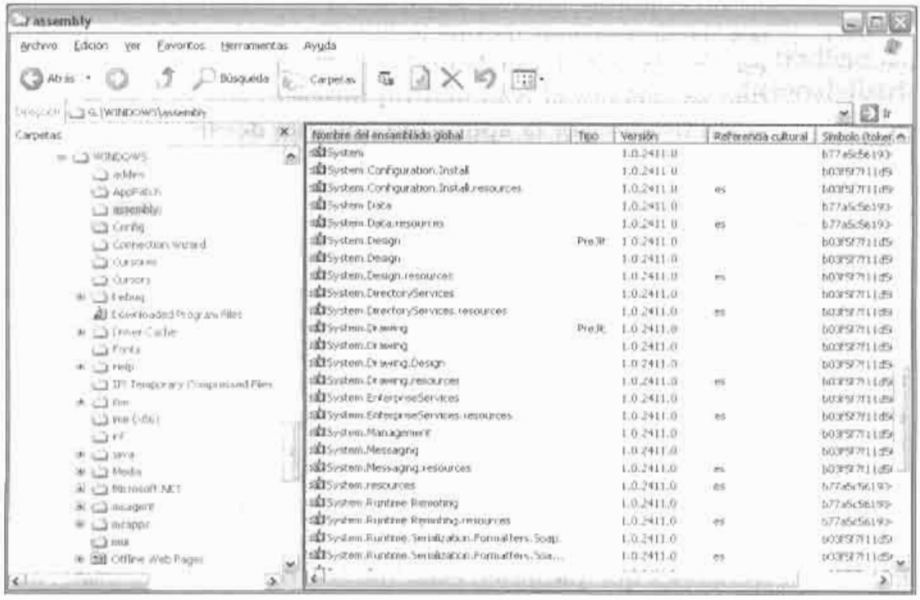


Figura 3.5. Carpeta en la que está el depósito de ensamblados compartidos

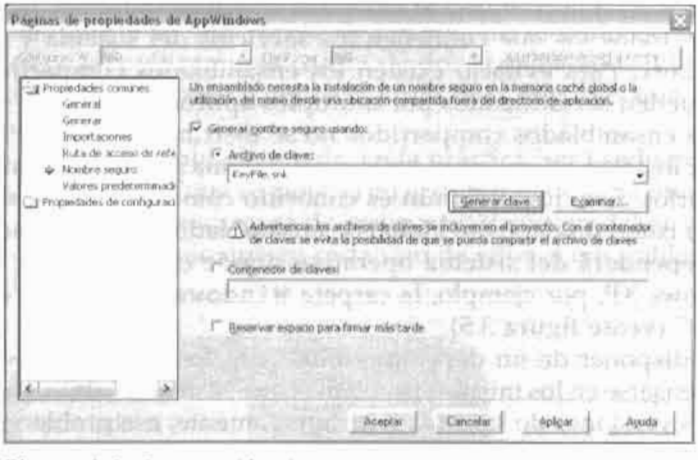


Figura 3.6. Generación de una nueva clave desde la ventana de opciones del proyecto

Proceso de compilación y ejecución

Ahora que ya sabemos, básicamente, cuál es la estructura de una aplicación .NET al nivel de módulos, ensamblados y contextos de ejecución, vamos a en-

trar con algo más de detalle en el proceso de ejecución del código que contienen esos módulos. Nos interesará conocer, especialmente, los pasos de verificación, compilación y ejecución en el entorno común de manera supervisada.

Las indicaciones dadas a continuación se aplican a todos los lenguajes .NET, ya que todos ellos generan sólo código MSIL y no código nativo. La excepción es Visual C++ .NET que, como posiblemente ya sabe, sí es capaz de generar código nativo Windows. A ese tipo de código no se aplicarían las normas que van a describirse.

Código intermedio

Actualmente la mayoría de las herramientas existentes para el desarrollo de aplicaciones generan código nativo, directamente ejecutable, para un determinado procesador y sistema operativo. Siempre se ha asumido que es la manera de obtener el mejor rendimiento, aunque esto dependería mucho del compilador utilizado. Algunos compiladores no se actualizan a medida que surgen nuevos procesadores, por lo que no aprovechan los nuevos conjuntos de instrucciones y posibilidades.

Una aplicación nativa no puede utilizarse directamente en otros sistemas operativos, en ocasiones ni en distintas versiones del mismo sistema, y tampoco se ejecutarían en otros procesadores. Es necesario, por lo tanto, actualizar y recompilar, contando con múltiples versiones de la misma aplicación: una para cada combinación de procesador y sistema operativo.

Mediante los lenguajes que forman parte de Visual Studio .NET, y sus respectivos compiladores, lo que obtenemos es siempre código MSIL o código intermedio. Es un código independiente de sistemas operativos y procesadores por lo que, en principio, nunca tendremos que recompilar un proyecto a causa de un cambio de procesador o de sistema. Es posible generar código MSIL con C# y Visual Basic .NET, pero también con multitud de lenguajes de terceros fabricantes, desde Perl hasta COBOL y Fortran, pasando por un ensamblador específico.

Lógicamente, el código MSIL no puede ser ejecutado directamente. Esta característica puede ser vista como un inconveniente o bien como una ventaja, dependiendo de cómo se mire. La misma aplicación .NET podría ejecutarse en Windows 2000, Windows CE o la próxima versión de Windows 64 bits, aprovechando en todos los casos los recursos del procesador y sistema sin, por ello, tener que compilar ni modificar nada.

Puede ver el código MSIL de un módulo con la misma herramienta `ildasm` utilizada previamente. Es algo que ya hizo en el primer capítulo, obteniendo la definición del método `Main()` en ensamblador MSIL.

El sistema común de tipos

Para generar código MSIL válido, que pueda ser ejecutado en el CLR, los compiladores deben hacer uso de los tipos y operaciones definidos en el CTS

(*Common Type System*), un sistema común en el que se recogen todos los tipos de datos y todas las operaciones que es posible efectuar, recogiendo la funcionalidad de un gran abanico de lenguajes.

Podríamos decir que en el CTS se recogen las instrucciones básicas así como los tipos de datos elementales que pueden utilizarse en el entorno común de ejecución.

No todos los lenguajes .NET harán uso de todas las posibilidades del CTS, usando tan sólo los elementos que requieran. No importa que el lenguaje de alto nivel utilizado sea BASIC, COBOL, C# o Pascal, el código para, por ejemplo, codificar un bucle será sintácticamente distinto en cada lenguaje, pero el código MSIL resultante será idéntico.

Partiendo de la base de que el código MSIL es indistinto del lenguaje de alto nivel usado para generarlo, es fácil comprender que un lenguaje puede utilizar código escrito en otro, por ejemplo en forma de componente, sin ningún problema. En realidad, esto será cierto siempre que ninguno de los lenguajes use elementos del CTS que el otro no comprende. Hay que tener en cuenta que el CTS recoge características y operaciones que pueden no estar contempladas en todos los lenguajes de alto nivel.

La sobrecarga de operadores en clases propias, por ejemplo, es una posibilidad contemplada en el CTS y posible en C# o C++, pero no en Visual Basic .NET.

Existe, no obstante, un mínimo común denominador, un subconjunto del CTS que sí existe en todos los lenguajes .NET. Ese subconjunto está recogido en la CLS (*Common Language Specification*), una especificación que define las operaciones y tipos que deben utilizarse si se quiere obtener un código que pueda cooperar sin limitaciones con otros lenguajes. Los ensamblados que contienen código ajustado a la CLS pueden ser usados desde otros proyectos aunque el lenguaje sea distinto. Es posible, por ejemplo, definir una clase en Visual Basic .NET y, posteriormente, derivar una nueva con C#. Esto no sería posible si no respetase la CLS.

Que un determinado ensamblado se ajuste a la CLS o no es un aspecto que se configura mediante un atributo. Éste, habitualmente, toma por defecto el valor `True`, que es lo que, por regla general, nos interesará. Podemos, no obstante, abrir el módulo `AssemblyInfo` y modificar dicho atributo tal como se muestra en la figura 3.7. De esta forma podríamos usar construcciones recogidas en el CTS pero no en la especificación común para hacer posible la interoperabilidad entre lenguajes.

Nota

Un proyecto puede utilizar internamente construcciones no ajustadas a la CLS pero no exponiéndolas al exterior, no haciéndolas públicas, obteniendo como resultado una aplicación ajustada a la CLS. Dicho de otra manera: para que una aplicación esté ajustada a la CLS basta con que los tipos expuestos públicamente cumplan con dicha especificación.

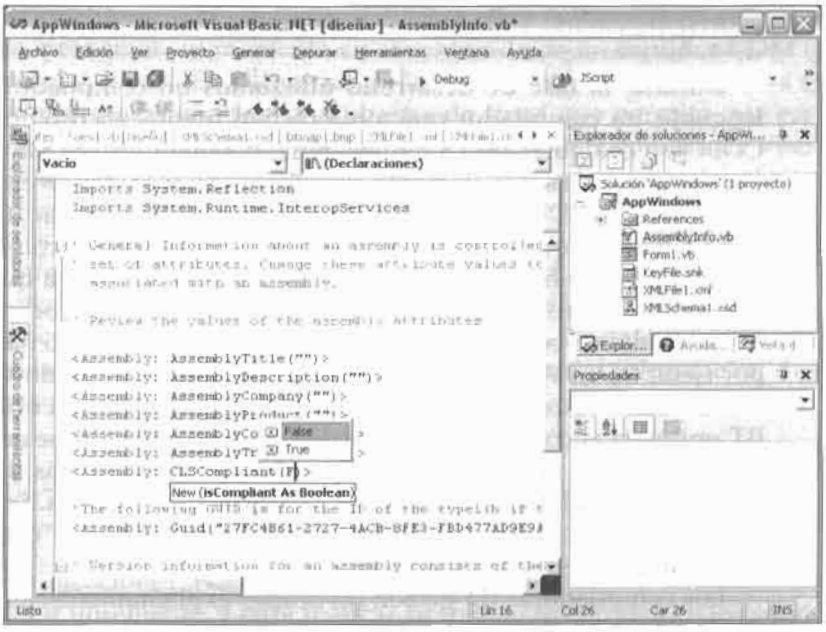


Figura 3.7. Modificamos el atributo `CLSCompliant` del ensamblado

Ejecución supervisada

El código MSIL, una vez compilado a código nativo mediante el compilador JIT, será ejecutado bajo la supervisión del CLR. En realidad, el código es verificado antes de compilarse como parte del proceso de iniciación de la aplicación. En esa verificación el CLR comprueba que el código MSIL no almacena construcciones peligrosas que no puedan ser supervisadas, tales como el uso de punteros.

Las diferentes tareas del CLR persiguen comprobar que el código puede ser ejecutado según los permisos establecidos, facilitar los servicios básicos de gestión de memoria o control de excepciones, facilitar la cooperación entre lenguajes o la ejecución distribuida o asegurar la ausencia de conflictos entre diferentes dominios de aplicación.

El código que se ejecuta bajo supervisión del CLR se denomina *managed code* o código gestionado. Esta supervisión hace posible, por ejemplo, que en los programas no tengamos que preocuparnos de la liberación de la memoria ocupada por los objetos, puesto que es el propio CLR, concretamente el recolector de basura, quien controla la vida de esos objetos y los destruye cuando no son necesarios.

Gracias a la ejecución supervisada de las aplicaciones, el CLR garantiza una estabilidad superior a la existente tradicionalmente en las aplicaciones Windows. Una aplicación .NET difícilmente puede ejecutarse conteniendo un error que provoque un fallo incontrolado, y mucho menos que afecte a otros dominios de aplicación o el propio sistema.

Compilación JIT

Durante la fase de desarrollo utilizamos un compilador dependiente del lenguaje, un compilador que toma código fuente, escrito en Visual Basic, C#, COBOL o cualquier otro lenguaje .NET, y genera código MSIL. Ese código, como ya sabe, no es directamente ejecutable. Aquí es donde entran en escena los compiladores JIT o *just in-time*, conocidos así por efectuar una compilación *al vuelo* a medida que es necesario. También se les conoce como *jitters*.

Observe que hablamos de compiladores, en plural, ya que existirán múltiples compiladores JIT según el sistema operativo y el procesador. De esta manera, un mismo código MSIL puede dar lugar a un código nativo distinto para, por ejemplo, un Windows 64 bits sobre Itanium y un Windows CE sobre Pocket PC. En unos casos, cuando el sistema es rápido y con recursos, el compilador JIT optimiza el código todo lo posible, aunque para ello se emplee más memoria y más ciclos del procesador. Cuando el entorno es más limitado, por ejemplo sobre dispositivos móviles, el *jitter* prácticamente se limita a traducir las sentencias MSIL a órdenes nativas, sin optimizaciones y consumiendo la menor cantidad de recursos posible.

Los compiladores JIT no son invocados explícitamente, sino que es el entorno de ejecución el que los usa según es necesario. Por regla general no se compila todo el código de los ensamblados a medida que éstos se alojan en memoria, sino que van compilándose individualmente los métodos a medida que son invocados. Esto reduce el tiempo de iniciación de la aplicación. Por otra parte, cada método es compilado una sola vez. Al invocar por primera vez a un método se llama al *jitter* y a continuación se ejecuta. Las veces siguientes es ejecutado directamente. De esta forma se obtiene un buen balance entre velocidad de puesta en marcha y velocidad de ejecución.

Cuando el código de un determinado ensamblado va a usarse siempre sobre la misma versión de la plataforma .NET y el mismo sistema operativo y procesador, no hay necesidad de efectuar la compilación dinámicamente, durante la ejecución, siendo posible compilar a código nativo. Para ello existe lo que se ha venido a llamar un *pre-jitter*, una herramienta que podemos utilizar desde la línea de comandos para compilar cualquier ensamblado. Esto es especialmente apropiado, por ejemplo, en el caso de los propios servicios de la plataforma .NET. Éstos se encuentran precompilados, por lo que no es necesario volver a compilar su código con cada aplicación.

Información de tipos

Aparte de código y recursos, los ensamblados también contienen información sobre los elementos o tipos con que cuentan. A esta información se le llama habitualmente *metadata* o meta-información. Gracias a la información de tipos, el CLR puede efectuar todo el trabajo que se ha descrito en un punto previo. El compilador JIT convierte el código MSIL en código nativo, pero deja la información de tipos en manos del CLR.

Para establecer una analogía, y comprender mejor qué es esta meta-información, pensemos un momento en el contenido de un servidor COM que aloja un control ActiveX. En el interior de la librería no sólo se encuentra el código del control, también existe una librería de tipos que permite que ese control pueda ser manipulado visualmente, por ejemplo en el entorno de Visual Basic. Visual Basic no podría mostrar las propiedades de los controles si no existiese esa librería de tipos. El caso de la información de tipos incluida en los ensamblados es idéntico.

La información de tipos en los ensamblados no sólo hará posible la manipulación visual de los componentes, sino que también será útil, por ejemplo, para trabajar con objetos mediante enlace en ejecución. Usando determinados servicios de la plataforma incluso podemos recuperar esa información desde aplicaciones propias.

Servicios .NET para las aplicaciones

Las aplicaciones .NET se escriben en un determinado lenguaje, que podemos elegir libremente, y para realizar su trabajo se apoyará en los servicios que ofrece la propia plataforma .NET. Estos servicios se alojan en ensamblados precompilados, ensamblados compartidos que podemos encontrar en el depósito global del que hablábamos anteriormente.

En los capítulos siguientes conoceremos algunos de los servicios .NET, parte de los cuales se han representado en la figura 3.8. Por una parte tenemos los servicios que podríamos denominar comunes, accesibles independientemente del tipo de aplicación que estemos desarrollando. Con ellos podemos trabajar con bases de datos, manipular documentos o XML, o acceder a elementos remotos. Sobre éstos se encuentran los servicios para la Web y para Windows.

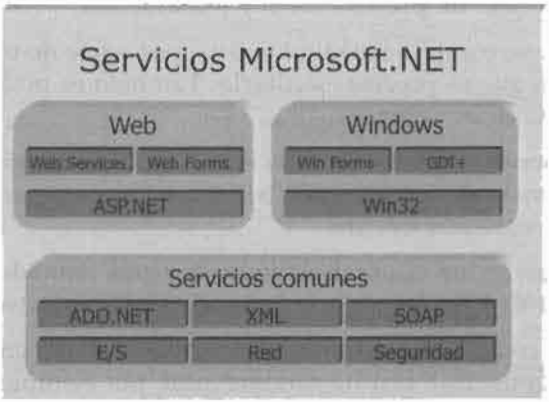


Figura 3.8. Esquema de parte de los servicios de la plataforma Microsoft.NET

Desde nuestras aplicaciones, usaremos estos servicios creando objetos de clases definidas en determinados ámbitos o espacios con nombre. Al crear, en

el segundo capítulo, un formulario Windows, por ejemplo, estábamos usando una clase, llamada `Form`, definida en el ámbito `System.Windows.Forms`.

Puntos clave

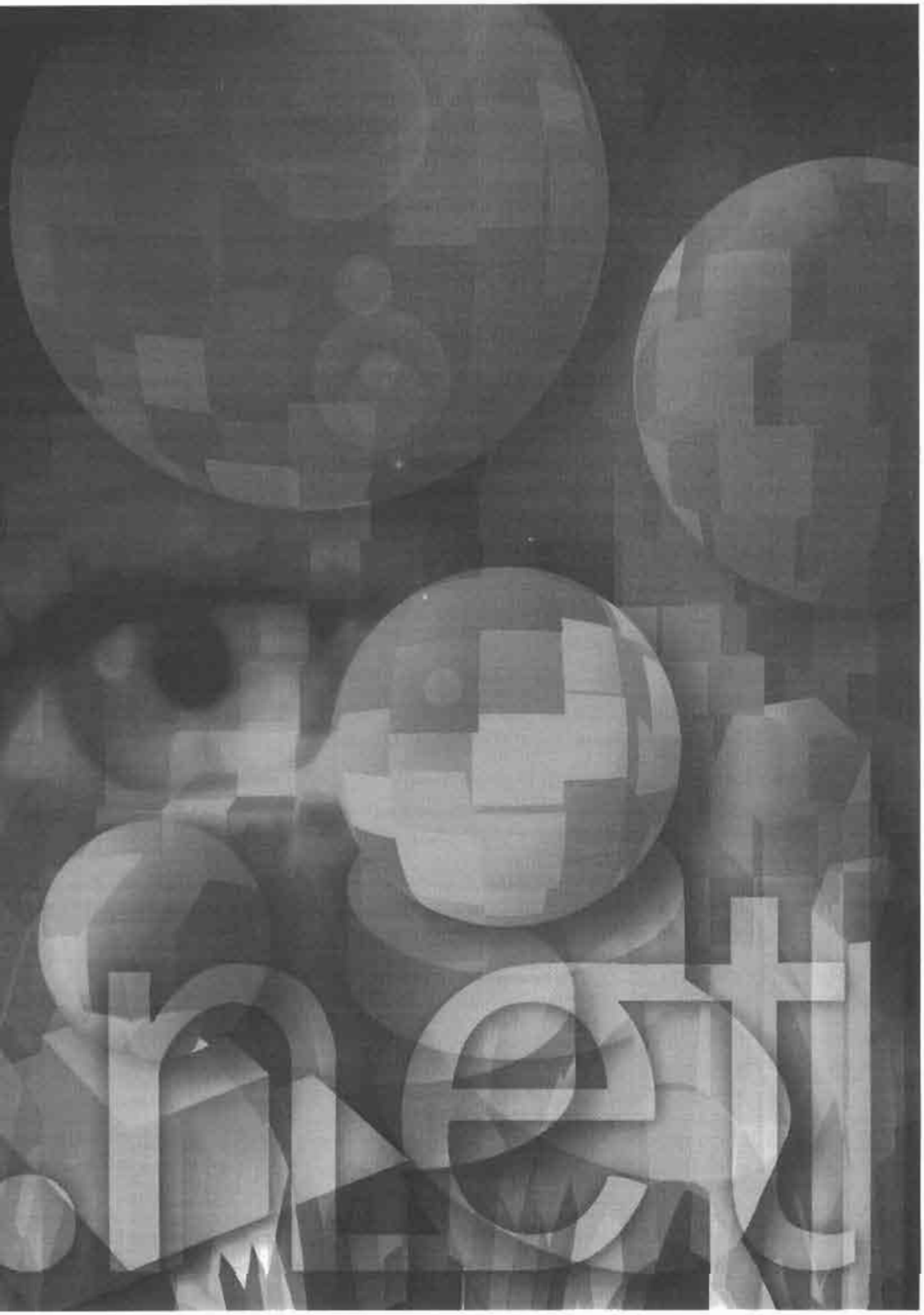
- Las necesidades de desarrollo actuales son muy distintas a las de hace unos años y, consecuentemente, las herramientas de desarrollo utilizadas hasta ahora comienzan a no ser suficientes.
- La plataforma Microsoft .NET facilita un entorno de ejecución común para múltiples lenguajes, un sistema de tipos también común y una biblioteca de clases con servicios fundamentales.
- El ensamblado es la unidad básica de distribución y seguridad de una aplicación .NET. Un ensamblado consta de un manifiesto, código IL y recursos.
- Un ensamblado puede constar de un solo módulo o estar formado por varios. El manifiesto, alojado en uno de los módulos, establece la relación lógica entre ellos que da lugar al ensamblado.
- La mayoría de los ensamblados son privados, pudiendo usarse tan sólo internamente en el proyecto del que forman parte. Existen ensamblados compartidos, como los que albergan los servicios de la plataforma .NET, que se alojan en un lugar bien conocido.
- Al iniciar una aplicación .NET, los ensamblados que la componen se alojan en el contexto de un dominio de aplicación.
- Al compilar un proyecto Visual Basic .NET el ensamblado obtenido no aloja código ejecutable, sino código intermedio. Éste, conocido como MSIL, es independiente de procesadores y plataformas.
- El código IL se compila a código nativo, por parte de un *jitter*, justo en el momento en que es preciso ejecutarlo. También es posible precompilar a código nativo si se desea.
- Las aplicaciones .NET se ejecutan en un entorno común, conocido como CLR, que consta de servicios básicos como los encargados de gestionar la memoria y recolectar basura.
- El CLR reconoce un conjunto común de tipos llamado CTS. El CLS, un subconjunto de CTS, asegura la interoperabilidad entre lenguajes.
- Además de código, los ensamblados también alojan en su interior información de tipos. Esta resulta fundamental, por ejemplo, a la hora de manipular los componentes en entornos visuales.
- La plataforma .NET consta de una serie de servicios, en forma de biblioteca de clases de componentes, que facilitan el desarrollo de todo tipo de aplicaciones: Windows, Web, servicios, acceso a datos, etc.

Resumen

Al finalizar este capítulo ya disponemos de una noción general sobre la estructura de las aplicaciones .NET, así como del proceso que se sigue hasta llegar a su ejecución. Partiendo de nuestro proyecto se generan uno o más ensamblados, cada uno de los cuales aloja código MSIL e información de tipos. Cada ensamblado puede componerse de múltiples archivos, utilizándose un manifiesto para definir tanto los elementos que forman parte del ensamblado como las dependencias que tiene respecto a otros.

Los ensamblados se alojan en memoria, en el momento de la ejecución, tras ser verificados por el CLR. Éste es el encargado de crear el dominio o `AppDomain` para la nueva aplicación, incluir en él los ensamblados y crear el primer hilo de ejecución. A medida que es necesario, el código MSIL va compilándose a código nativo, trabajo que efectúa el compilador JIT o *jitter*. El CLR, aparte de ofrecer servicios básicos, supervisa la ejecución encargándose de tareas como la recolección de basura o gestión de excepciones.

Se han introducido muchos términos y conceptos que, a partir del capítulo siguiente, necesitaremos conocer para ir avanzando en el conocimiento del lenguaje Visual Basic .NET y los servicios .NET.



4
Tipos de datos

Un programa es una secuencia de sentencias u órdenes cuyo fin principal, generalmente, es manipular información. Esa información se almacena en variables, se introduce en el código en forma de constantes o es recuperada de diversas fuentes. En cualquier caso es fundamental que conozcamos los tipos de datos que existen en Visual Basic .NET, así como la forma de aprovecharlos en nuestros propios programas.

El objetivo de este capítulo no es sólo darle a conocer los tipos de datos disponibles en el lenguaje, sino también mostrarle cómo declarar y usar variables, crear sus propias estructuras de datos o cómo trabajar con arreglos, caracteres y constantes. Los cambios respecto a Visual Basic 6, si es que conocía esa versión del lenguaje, son importantes. Sólo para abrir boca, sepa que en Visual Basic .NET no existe el tipo `Variant` y los nuevos tipos de datos se definen como estructuras.

El sistema común de tipos

En el capítulo anterior se citaba la existencia de CTS, un sistema común de tipos compartido por todos los lenguajes disponibles para la plataforma Microsoft .NET. Visual Basic .NET, como lenguaje ajustado a la especificación CLS, usa parte de los tipos del CTS. Éstos pueden clasificarse según el esquema de la figura 4.1. Como puede ver existen dos grandes ramas: los tipos que almacenan valores y los que alojan referencias. Es una diferencia fundamental.

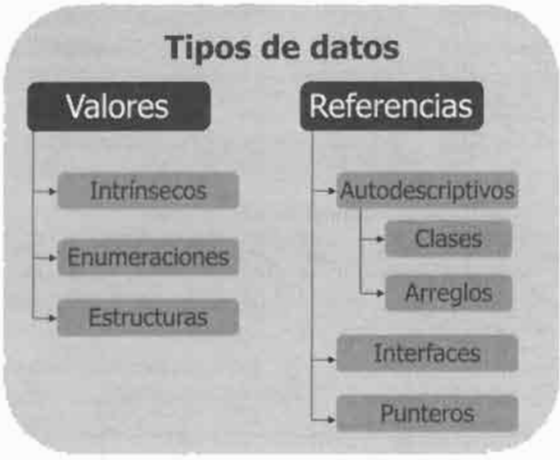


Figura 4.1. Clasificación de los tipos de datos .NET

Los tipos de datos que almacenan directamente un valor ocupan un espacio de memoria en la pila, generalmente asignado de manera estática, de tal forma que su manipulación resulta muy eficiente. Por el contrario, los representados en la rama derecha son tipos de datos que podríamos denominar como complejos. El espacio de memoria necesario para almacenar la información es asignado dinámicamente, tomándolo de un área conocido habitualmente como *heap*. Lo que se obtiene es una referencia al lugar en el que se encuentra el valor, no el valor en sí. Cada vez que quiere recuperarse al valor, o manipularse, es necesario tomar la referencia y acceder al espacio de memoria previamente asignado. El tiempo empleado, por tanto, es algo mayor.

En este capítulo vamos a centrarnos básicamente en los tipos de datos de la rama izquierda: intrínsecos, enumeraciones y estructuras. Posteriormente abordaremos el tratamiento de los distintos tipos de clases y las interfaces. Si trataremos a continuación el uso de arreglos.

Un sistema de tipos unificado

La mayoría de lenguajes actuales, excepción hecha de Smalltalk y pocos más, diferencian claramente entre objetos y datos que no son objetos. En C++, por poner un ejemplo bien conocido, los tipos de datos intrínsecos, como enteros y caracteres, no pueden ser tratados como objetos de una clase. Podríamos, no obstante, definir clases que encapsulan esos tipos de datos, eso sí, a costa de una bajada importante del rendimiento, que es lo que ocurre en Smalltalk.

Visual Basic .NET cuenta con un sistema de tipos unificado y combina lo mejor de ambos mundos, de los lenguajes en los que todo es un objeto y de aquellos en los que existen tipos de datos que no son objetos. Para conseguir esto todos los tipos de datos de Visual Basic .NET parten de una raíz común, llamada `Object`, de la que derivan, por una parte, los tipos que almacenan directamente el valor y, por otra, los que mantienen referencias.

En la figura 4.2 se ha representado esquemáticamente, y de manera simplificada, este sistema de tipos unificado. La raíz, como acaba de decirse, es `Object`. Uno de sus tipos derivados es `ValueType` que, como puede ver, actúa como base de todos los tipos de datos intrínsecos, como `Byte`, `Char` o `Integer`, así como de las enumeraciones y estructuras definidas por el usuario.

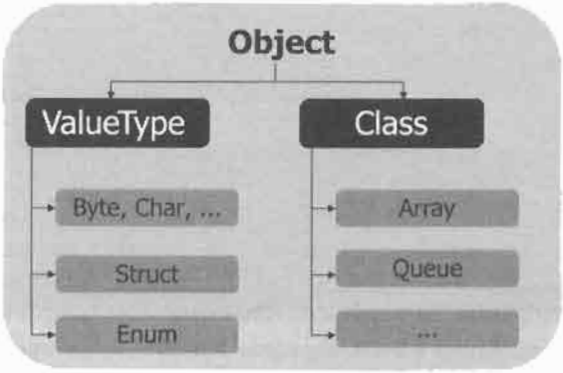


Figura 4.2. Sistema de tipos unificado de Visual Basic .NET

La rama de la derecha representa cualquier clase, ya sea existente en la plataforma .NET, como `Array` y `Queue`, o definida por el usuario. Todas ellas derivan también de `Object`. Contamos, por lo tanto, con un ascendiente común para todos los tipos, independientemente de que almacenen un valor o una referencia. Esto tiene muchas ventajas puesto que, en la práctica, cualquier tipo puede ser tratado como si fuera un objeto, al estilo de Smalltalk. Los tipos que almacenan valor, sin embargo, tienen un rendimiento muy eficiente, equiparable a los tipos intrínsecos de C++ y muchos otros lenguajes.

Empaquetado y desempaquetado de datos

El almacenamiento en memoria de un tipo de dato por valor y otro por referencia, representado en la figura 4.3, afecta al rendimiento y, lógicamente, determina en cierta manera la forma de utilizar el dato en sí. La variable entera que aparece en esa figura contiene directamente un valor, de tal forma que el acceso al dato es directo y prima el rendimiento. La representada debajo, una variable que tiene como tipo una clase, almacena una referencia. El acceso al dato requiere un paso adicional, pero a través de esa referencia puede accederse no sólo al valor, sino a todos los miembros de que disponga de la clase: métodos, propiedades, etc.

La existencia de una raíz común, según se ha visto en el punto previo, facilita la conversión de cualquier tipo de dato que almacena un valor en un objeto que lo contiene, y viceversa. Es lo que se conoce como *boxing* (empaquetado) y *unboxing* (desempaquetado). El entero que aparece en la figura 4.3, por ejemplo, puede ser empaquetado para tratarlo como un objeto en una colección de objetos heterogéneos, algo que no podríamos hacer si fuese un valor simple.

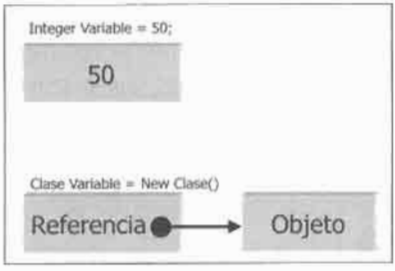


Figura 4.3. Representación de dos variables, una conteniendo un valor y otra una referencia

Dado que tanto los tipos que almacenan valores como los que contienen referencias son derivados de `Object`, la conversión de uno a otro, empaquetado o desempaquetado, se produce automáticamente, de manera implícita, aunque también podemos especificar de manera explícita el tipo deseado. El código siguiente muestra algunas situaciones en los que se empaqueta y desempaqueta un valor entero.

```
Dim Numero As Integer = 5

Dim RNumero As Object

RNumero = Numero

Console.WriteLine(RNumero)

RNumero = 10

Numero = CInt(RNumero)

Console.WriteLine(Numero)
Console.WriteLine(RNumero)

Console.WriteLine(10.GetType())
```

Nota

Observe en este código, en la declaración de variables, que es posible asignar un valor inicial en el mismo momento de la declaración. Es una de las novedades de Visual Basic .NET que contribuye a simplificar el trabajo, al no ser necesaria una asignación separada tras cada declaración.

Al ejecutar esta aplicación de consola obtenemos el resultado que puede verse en la figura 4.4. En principio `RNumero` toma el valor que tenía inicialmente `Numero` en el momento de proceder al empaquetamiento. Observe, ya al final del código, cómo podemos incluso usar una constante para invocar a métodos

de `Object`, en este caso concreto en el método `GetType()`. En ese caso también se produce un empaquetado del valor para obtener un objeto, tras lo cual se llama a `GetType()`.

Figura 4.4. Valores empaquetados y desempaquetados

Recurriendo una vez más a la herramienta `ildasm`, que hemos usado en capítulos previos, podemos acceder al código MSIL generado a partir del programa anterior y ver, como en la figura 4.5, el momento en que se realiza el empaquetado y desempaquetado de datos. En dicha figura, por ejemplo, se ha destacado la línea en la cual se empaqueta el valor `10`, que aparece en la posición anterior, justo antes de llamar al método `GetType()`.

Tipos intrínsecos

Conocemos con esta denominación a los tipos de datos que podríamos considerar como fundamentales para el lenguaje: números enteros y con parte decimal de distintos tamaños, caracteres, cadenas, lógicos, etc. Si ha programado previamente con Visual Basic posiblemente conozca tipos como `Integer`, `Long` o `Boolean`, pero seguramente no sabrá que algunos de esos tipos, aunque conservan el nombre, son distintos, así como la existencia de otros nuevos: `Char`, `Decimal` y `Object`.

La tabla 4.1 resume los tipos de datos de Visual Basic .NET. En la plataforma .NET existen otros, parte del CTS pero no de la especificación CLS, que no existen como tales en Visual Basic .NET pero sí en otros lenguajes, tales como C# o C++.

Tabla 4.1. Tipos de datos intrínsecos de Visual Basic .NET

Tipo	Dato que puede contener
`Byte`	Número entero de 8 bits sin signo
`Short`	Número entero de 16 bits sin signo
`Integer`	Número entero de 32 bits sin signo

Tipo	Dato que puede contener
`Long`	Número entero de 64 bits sin signo
`Single`	Número con parte decimal de precisión simple
`Double`	Número con parte decimal de precisión doble
`Decimal`	Número con parte decimal fija para cálculos financieros
`Boolean`	`True` o `False`
`Char`	Un carácter del conjunto Unicode
`String`	Cadena de caracteres
`Object`	Referencia a cualquier objeto

```
// Module1::Main : void()
.method public static void  Main() cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
  // Code size       76 (0x4c)
  .maxstack  1
  .locals init ([0] int32 Numero,
           [1] object BNumero)
  IL_0000:  nop
  IL_0001:  ldc.i4.5
  IL_0002:  stloc.0
  IL_0003:  ldloc.0
  IL_0004:  box        [mscorlib]System.Int32
  IL_0009:  stloc.1
  IL_000a:  ldloc.1
  IL_000b:  call       object [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::GetObj
  IL_0010:  call       void [mscorlib]System.Console::WriteLine(object)
  IL_0015:  nop
  IL_0016:  ldc.i4.s   10
  IL_0018:  box        [mscorlib]System.Int32
  IL_001d:  stloc.1
  IL_001e:  ldloc.1
  IL_001f:  call       int32 [Microsoft.VisualBasic]Microsoft.VisualBasic.Helpers.IntegerType::
  IL_0024:  stloc.0
  IL_0025:  ldloc.0
  IL_0026:  call       void [mscorlib]System.Console::WriteLine(int32)
  IL_002b:  nop
  IL_002c:  ldloc.1
  IL_002d:  call       object [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::GetObj
  IL_0032:  call       void [mscorlib]System.Console::WriteLine(object)
  IL_0037:  nop
  IL_0038:  ldc.i4.s   10
  IL_003a:  box        [mscorlib]System.Int32
  IL_003f:  callvirt   instance class [mscorlib]System.Type [mscorlib]System.Object::GetType()
  IL_0044:  call       void [mscorlib]System.Console::WriteLine(object)
  IL_0049:  nop
  IL_004a:  nop
  IL_004b:  ret
} // end of method Module1::Main
```

Figura 4.5. En el código MSIL es fácil encontrar el punto en el que se empaquetan los datos

Observe que ya no existen tipos como `Currency` o `Variant` y que el tamaño de los tipos enteros ha cambiado. Ahora `Integer` es un número de 32 bits, no de 16 como en Visual Basic 6, y `Long` es un entero de 64 bits, tipo inexistente en versiones previas del lenguaje.

Cada uno de los tipos de Visual Basic tiene una correspondencia con un tipo del CTS de la plataforma .NET. En realidad, los tipos de la tabla 4.1 es el nombre que se les ha dado en Visual Basic .NET a algunos de los tipos del CTS. En la tabla 4.2 puede ver la correspondencia entre tipos de Visual Basic .NET y su

denominación en el ámbito `System` de la plataforma .NET. Al final de la tabla aparecen los tipos que no tienen un equivalente en Visual Basic .NET.

Tabla 4.2. Equivalencia de tipos entre Visual Basic .NET y el CTS

Tipo CTS	Tipo Visual Basic .NET
`System.Byte`	`Byte`
`System.Int16`	`Short`
`System.Int32`	`Integer`
`System.Int64`	`Long`
`System.Single`	`Single`
`System.Double`	`Double`
`System.Decimal`	`Decimal`
`System.Boolean`	`Boolean`
`System.Char`	`Char`
`System.String`	`String`
`System.Object`	`Object`
`System.SByte`	No existe
`System.UInt16`	No existe
`System.UInt32`	No existe
`System.UInt64`	No existe
`System.IntPtr`	No existe
`System.UIntPtr`	No existe

Tipos no ajustados a la CLS

Los tipos de datos que aparecen al final de la tabla 4.2 forman parte del sistema común de tipos de la plataforma .NET, por lo que cualquiera que desarrolle un nuevo lenguaje puede, opcionalmente, implementar su uso. No son tipos de datos, sin embargo, que aparezcan en la especificación CLS. Esto significa que su presencia en un lenguaje no es indispensable, por eso en Visual Basic .NET, concretamente, no existen como tipos nativos.

Que esos tipos no aparezcan como nativos en el lenguaje no significa que no puedan ser usados. Al ser parte de la plataforma .NET, es posible declarar variables de esos como si fuesen estructuras creadas por nosotros mismos. En el siguiente programa, por ejemplo, se muestra cómo utilizar el tipo `IntPtr` para trabajar con un puntero. Se usan los servicios de interoperabilidad de la plataforma .NET para asignar un bloque de memoria, escribir un dato en él, posteriormente recuperarlo y, finalmente, liberar esa memoria.

```
Imports System.Runtime.InteropServices
Module Module1

  Sub Main()

    Dim Puntero As System.IntPtr

    Puntero = Marshal.AllocHGlobal(255)

    Marshal.WriteByte(Puntero, 16)

    Console.WriteLine("{0} = {1}", Puntero.ToString(), _
       Marshal.ReadByte(Puntero))

    Marshal.FreeHGlobal(Puntero)
  End Sub

End Module
```

Nota

La sintaxis `{0}` y `{1}` en el primer parámetro del método `WriteLine()` se utiliza para introducir los valores de los parámetros que se facilitan en segundo y tercer lugar.

La ejecución del programa genera la salida de una cadena de caracteres (véase la figura 4.6) en la que se muestra la dirección del bloque de memoria asignado, recuperada directamente a partir de la variable `IntPtr` mediante el método `ToString()`, y el valor recuperado de esa dirección.

Figura 4.6. Dirección del bloque de memoria y valor que contiene

De manera análoga podríamos usar los demás tipos de datos que, aunque no son nativos de Visual Basic .NET, sí forman parte de la plataforma .NET.

Declaración de variables

Los tipos de datos no son útiles por sí mismos, aisladamente. Su finalidad es facilitar la manipulación de información, por ejemplo estableciendo la estructura de una variable en memoria y el formato de los datos que puede contener. Las variables podemos declararlas en distintos ámbitos: interior de una función, interior de una clase, con ámbito de módulo, etc. Además, existen una serie de palabras clave que nos permiten indicar la visibilidad de la variable fuera de ese ámbito.

Además del ámbito, en la declaración también se establece el tipo de las variables que, como ya sabe, determina los valores que podrán contener. Opcionalmente, es posible dar un valor inicial en el mismo momento de la declaración. De no especificarse lo contrario, las nuevas variables tendrán asignado un valor por defecto.

Ámbitos y visibilidad

Utilizando la palabra clave `Dim` para efectuar la declaración, que es lo más habitual en versiones previas de Visual Basic, la variable tendrá el ámbito por defecto. Éste determina que una variable declarada en el interior de una función, por ejemplo, sólo sea visible en su interior y, además, su vida se limite al tiempo durante el que esa función está ejecutándose. Es lo que se conoce habitualmente como variable local.

Un ámbito aún más reducido tienen aquellas variables declaradas en un bloque de código, como puede ser un bucle o un condicional. Tan sólo puede accederse a ellas desde instrucciones existentes en ese bloque. En contraposición, una variable declarada fuera de cualquier función, al nivel de módulo o clase, será compartida por todo el código de esa entidad. Si además usamos el modificador `Public`, la variable será entonces visible en todo el *namespace*, el ámbito con nombre en el que se introducen los módulos y clases que formarán la aplicación.

Es más fácil comprender esto mediante algo de código. Iniciamos una nueva aplicación de consola e introducimos el código siguiente:

```
Module Module1

  Public Entero1 As Integer = 5

  Private Entero2 As Integer = 10

  Sub Main()

    Dim Entero3 As Integer = 15
```

```
    Auxiliar()


    Dim MiSimple As New Simple()
    MiSimple.Enumera()


    Console.WriteLine("Main()")
    Console.WriteLine("{0}, {1} y {2}", Entero1, Entero2, Entero3)
  End Sub

  Sub Auxiliar()
    Console.WriteLine("Auxiliar")


    Console.WriteLine("{0}, {1} y {2}", Entero1, Entero2, Entero3)
  End Sub
End Module


Class Simple

  Private Entero4 As Integer = 20

  Sub Enumera()
    Console.WriteLine("Simple.Enumera()")


    Console.WriteLine("{0}, {1} y {2}", Entero1, Entero4, Entero2)
  End Sub
End Class
```

Como puede ver, tenemos cuatro variables enteras llamadas `Entero1`, `Entero2`, `Entero3` y `Entero4`, cada una de ellas con un ámbito distinto. La primera está declarada en el interior de `Module1` pero usando el modificador `Public`, lo que da como resultado que la variable sea visible en todo el ámbito con nombre, o *namespace*, establecido como principal en la ventana de opciones del proyecto. La segunda, a pesar de declararse en el mismo punto, sólo es visible en `Module1` ya que se ha usado el modificador `Private`.

`Entero3` es una variable local al método `Main()`, de ahí que no sea accesible desde ningún punto externo, ni desde la función `Auxiliar()`, que está en el mismo módulo, ni desde una clase externa al módulo pero dentro del mismo *namespace*. Por último tenemos `Entero4`, una variable con visibilidad limitada a la clase `Simple`. Esto significa que cualquier método de esa clase puede acceder a la variable, pero ningún código externo tiene esa posibilidad.

El código, tal y como está mostrado más arriba, genera errores de compilación que, gracias a la compilación en segundo plano de Visual Basic .NET, son resaltados en el editor sin necesidad siquiera de llegar a compilar. Para obtener un ejecutable válido sustituya los parámetros no accesibles por el valor 0. Entonces sí podrá ejecutar, obteniendo un resultado similar al de la figura 4.7.

Figura 4.7. Valores de las variables en distintos puntos

Nota

No se preocupe en este momento por la sintaxis utilizada para definir una clase, crear un objeto a partir de ella y usar sus métodos. Es un aspecto del que nos ocuparemos en profundidad en un capítulo posterior.

Vida de una variable

Al trabajar con variables hay que distinguir entre lo que es la visibilidad, los puntos desde los que puede accederse a ella, y la vida de la variable, que es el tiempo transcurrido desde el momento en que empieza a existir hasta que se destruye. En el programa anterior, por ejemplo, la variable `Entero3` comienza a existir cuando se inicia la ejecución del método `Main()`, destruyéndose cuando éste finaliza.

Lo mismo ocurriría, por ejemplo, si hubiésemos declarado una variable en el método `Auxiliar()`. El tiempo de vida, por tanto, está limitado al espacio de tiempo durante el que se ejecuta el método en cuestión.

Distinto es el caso de la variable `Entero4`, declarada como miembro de una clase. Cada vez que se crea un objeto de esta clase se inicia la vida de la variable, permaneciendo accesible para todos los métodos de la clase hasta que el objeto sea destruido.

Por último tenemos las variables `Entero1` y `Entero2`. Ambas existen durante toda la ejecución del programa, desde que éste se inicia hasta que finaliza, si bien su visibilidad es distinta como se ha visto en el ejemplo previo.

Mediante ciertos modificadores es posible alterar el tiempo de vida de una variable sin, por ello, alterar su visibilidad. Por ejemplo, en el caso de la variable `Entero3`, local al método `Main()`, bastaría con cambiar `Dim` por `Static` para tener una variable que existe desde que se inicia el programa hasta que termina, es decir, con un tiempo de vida idéntico al de `Entero1` y `Entero2`. La visibilidad, sin embargo, seguiría siendo la misma. Una variable estática, nombre con el que se conoce a las declaradas usando la palabra `Static`, tiene

utilidad en caso de que deseemos mantener su valor más allá del tiempo de vida del bloque en que se encuentra.

Seguramente comprenderá el concepto de variable estática, y su funcionamiento, mucho mejor con un ejemplo. Nuevamente partimos de una aplicación de consola. En este caso además del método `Main()` contaremos con otro, llamado `MuestraVariables()`, en cuyo interior declararemos dos variables locales del mismo tipo. Estas dos variables no serán visibles fuera de dicho método pero, como puede ver en el código siguiente, su tiempo de vida será distinto. La primera se ha declarado con la palabra clave `Dim`, dando lugar a una variable *automática*. Así se conoce a las variables que son creadas cada vez que se entra en su ámbito y destruidas al salir de él. La segunda, por el contrario, es estática, de tal forma que existe durante toda la ejecución del programa.

```
Sub Main()
  Dim Contador As Byte


  For Contador = 1 To 10
    MuestraVariables()
  Next
End Sub


Sub MuestraVariables()

  Dim Valor1 As Byte

  Static Valor2 As Byte


  Valor1 += 1
  Valor2 += 1


  Console.WriteLine("Valor1={0}, Valor2={1}", Valor1, Valor2)
End Sub
```

Al ejecutar el programa obtendrá un resultado como el de la figura 4.8, en el que se ve claramente que en cada llamada al método `MuestraVariables()` la variable `Valor1` tiene de partida el valor `0`, mientras que `Valor2` conserva el contenido de la invocación anterior y, por ello, su valor va incrementándose en cada llamada.

Nota

Las variables declaradas en el interior de una clase pueden tener un tiempo de vida asociado a los objetos creados a partir de ella, según se ha descrito anteriormente, o una vida global asociada a la propia clase. Para ello se utiliza el modificador `Shared`. Es un tema del que nos ocuparemos en un capítulo posterior.

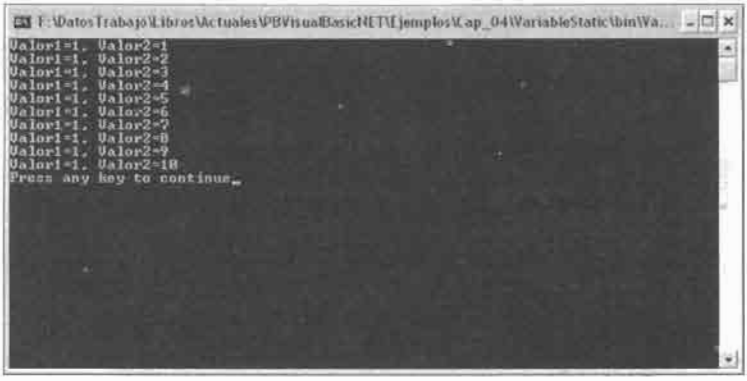


Figura 4.8. Valores obtenidos por las variables locales

Especificación de tipo

Además del ámbito, el tiempo de vida y el identificador de la propia variable, una declaración debe completarse especificando su tipo. Ya lo ha visto en los ejemplos previos, en los que hemos utilizado variables de tipo `Integer` y `Byte`. Básicamente, basta con disponer detrás del identificador de la variable la palabra `As` y el nombre del tipo que deseamos. En el caso de que múltiples variables vayan a compartir el mismo tipo, no es necesario indicarlo tras cada identificador, siendo suficiente la separación de identificadores mediante comas y la indicación del tipo al final, como se muestra en el siguiente ejemplo:

```
Dim Var1, Var2, Var3 As Integer, Var4, Var5 As Byte
```

En este caso `Var1`, `Var2` y `Var3` son de tipo `Integer`, mientras que `Var4` y `Var5` son de tipo `Byte`. Esto es un cambio importante respecto a lo que ocurría en versiones previas de Visual Basic en las que, al no haberse indicado un tipo justo detrás, `Var1`, `Var2` y `Var4` serían de tipo `Variant`.

Asignación de un valor inicial

Otra posibilidad nueva, inexistente en versiones previas de Visual Basic, es la asignación de un valor inicial a las variables en el mismo momento de la declaración, ahorrando así una asignación posterior. Es algo que ha podido ver también en alguno de los ejemplos usados en este capítulo. Basta con poner el símbolo = al final, seguido del valor inicial que, lógicamente, deberá estar en consonancia con el tipo que se ha indicado.

En caso de que estemos declarando más de una variable del mismo tipo, los valores de cada una de ellas irían detrás del identificador a excepción del correspondiente a la última, que iría detrás del tipo. En la línea siguiente tiene un ejemplo de este caso concreto.

```
Dim Var1 = 5, Var2 = 10, Var3 As Integer = 15
```

Además de valores constantes, en la asignación del valor inicial también podemos usar otras variables e, incluso, expresiones que den como resultado un valor del tipo que espera la variable.

Tipos definidos por el usuario

Además de los tipos de datos que podríamos denominar intrínsecos, predefinidos en el lenguaje, Visual Basic .NET nos permite utilizar cualesquiera otros que hayamos podido definir. Siempre ciñéndonos a los tipos que almacenan valores, no referencias, es posible crear básicamente dos tipos de datos: enumeraciones y estructuras. En un capítulo posterior veremos que también es posible definir tipos de referencia, concretamente clases de objetos.

Tanto las enumeraciones como las estructuras son entidades derivadas de `System.ValueType`. Una enumeración puede definirse prácticamente en cualquier lugar, como las declaraciones de variables, mientras que las estructuras sólo pueden aparecer en módulos y clases, siendo tipos más complejos que las enumeraciones.

Lógicamente, en Visual Basic .NET existen enumeraciones y estructuras predefinidas, muchas de las cuales conocerá al estudiar servicios como GDI+, en capítulos posteriores. Aparte, nosotros podemos también definir nuestros propios tipos.

Enumeraciones

La mayoría de los tipos de datos intrínsecos permiten que una variable tome un valor de una lista más o menos limitada. Una variable de tipo `Byte`, por ejemplo, puede contener un número entre 0 y 255, un `Boolean` sólo puede tener los valores `True` o `False`, y lo mismo ocurre con los tipos enteros y el tipo `Char`.

Siempre que necesitemos una variable que deba tomar un valor de una lista limitada, no existente en ninguno de los tipos intrínsecos ni enumeraciones predefinidos, podemos crear nuestra propia enumeración. Suponiendo que en una aplicación médica necesitemos variables que contenga valores representando músculos, huesos y otros elementos, podríamos definir las enumeraciones apropiadas. Una de ellas podría ser la siguiente:

```
Enum Dedo
  Pulgar
  Índice
  Corazón
  Anular
  Meñique
End Enum
```

El identificador `Dedo` es, a partir de esta definición, un nuevo tipo de dato. Podemos utilizarlo como cualquier otro, `Integer`, `String`, `Byte`, etc., para

declarar variables. Las variables de tipo `Dato` tan sólo podrán contener uno de los valores que aparecen en la enumeración. El editor de Visual Basic .NET, al efectuar una asignación de una variable de este tipo, se ocupa de mostrar (véase la figura 4.9) la lista de posibles valores.

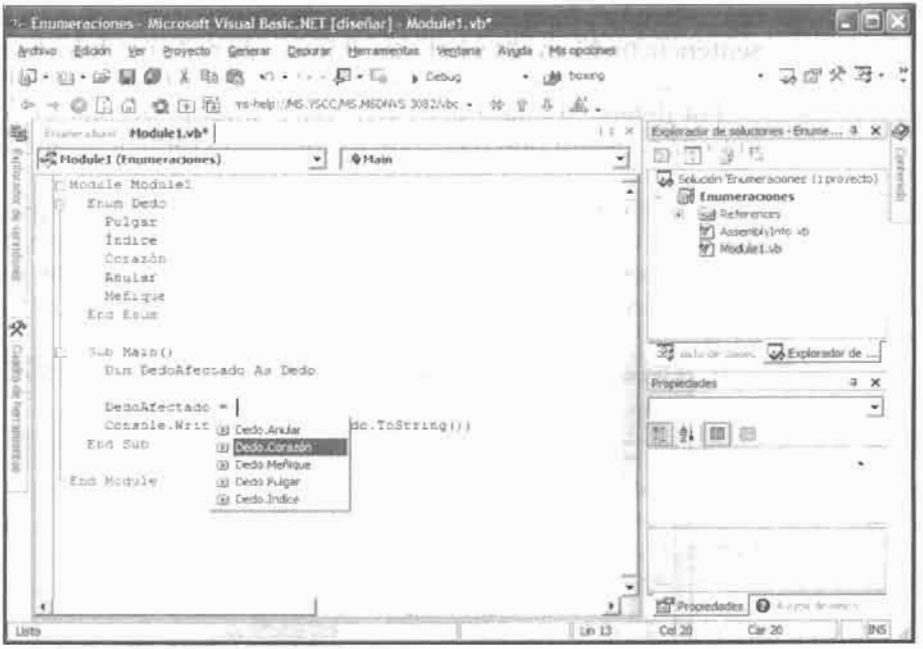


Figura 4.9. El editor reconoce el tipo de la variable y muestra los valores que podemos asignarle

Internamente las variables de tipo `Dedo` almacenan un valor entero que representa a la constante que corresponda. En el caso de la enumeración `Dedo`, la constante `Pulgar` tiene asociado el valor `0`, `Índice` el `1` y así sucesivamente. Puede comprobarlo fácilmente haciendo algo así:

```
DedoAfectado = Dedo.Corazón
Console.WriteLine(DedoAfectado)
```

Si lo que quiere obtener no es el valor equivalente a `Dedo.Corazón`, en este caso, sino el nombre de la constante, puede usar el método `ToString()` con el que cuentan la mayoría de los tipos de datos. Utilizando `DedoAfectado.ToString()`, por tanto, no obtendríamos el valor `2` sino la cadena `Corazón`.

Comprobación estricta de tipos

Si, partiendo de la enumeración definida en el punto anterior, declara una variable de tipo `Dedo` y después le asigna un valor numérico, comprendido entre 0 y 4, verá que Visual Basic .NET no genera ningún error. Lo que ocurre es

que se efectúa una conversión implícita, sin necesidad de que la indiquemos, convirtiendo el entero en la constante que corresponda.

Estas conversiones implícitas no sólo se dan en este caso, sino en algunos más y, en ocasiones, pueden inducir a errores cuando se obtiene un resultado que no era el que se esperaba. Para evitarlo, lo mejor es activar la comprobación estricta de tipos de Visual Basic .NET. Para ello basta con introducir la sentencia `Option Strict On` al inicio de cada módulo de código. Al hacerlo, una asignación como `Dedo = 2` generaría un error de compilación.

Por defecto Visual Basic .NET está configurado con `Option Strict` en estado `Off`, de ahí que haya que introducir la sentencia anterior para modificar el estado y activar la comprobación estricta de tipos.

También puede optar por acceder a las opciones del proyecto, como se ha hecho en la figura 4.10, y modificar ese estado por defecto. De esta manera no tendrá que introducir manualmente `Option Strict On` en cada módulo. Es lo más recomendable.

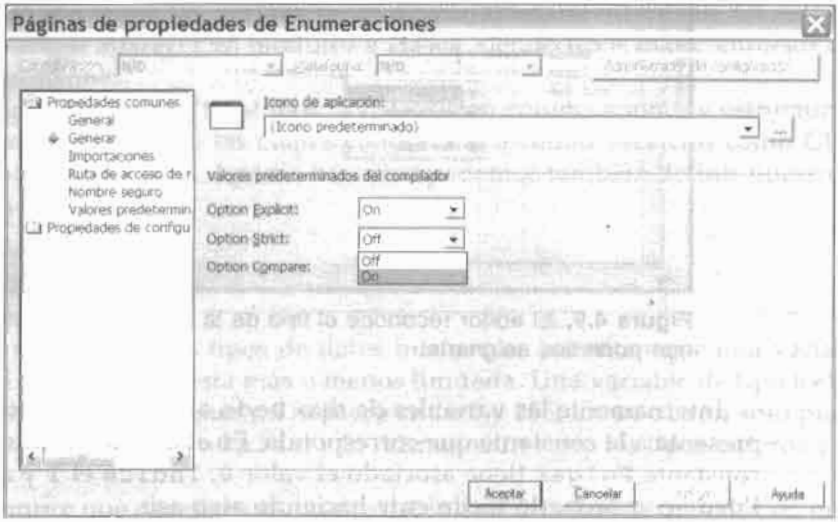


Figura 4.10. Alteramos el estado por defecto de `Option Strict`

La comprobación estricta de tipos afecta, como se ha dicho, no sólo a las enumeraciones. Al activarla tampoco podremos asignar directamente, con conversión implícita, un número decimal a una variable entera, por poner un caso. Es la situación que puede ver en la figura 4.10, en la que Visual Basic .NET nos informa sobre el error de la asignación directa. En estos casos tendremos que indicar, de manera explícita, que deseamos efectuar la conversión. Para ello utilizaremos funciones como `CInt()`, `CDbl()`, `CLng()`, etc., según deseemos convertir a `Integer`, `Double`, `Long`, etc. En el caso de la figura 4.11 tendríamos que utilizar `CInt(Pi)`.

Existen funciones de conversión para todos los tipos intrínsecos, así como una función genérica que nos permite convertir cualquier valor de un tipo a otro. Esta función, llamada `CType()`, necesita dos parámetros: el valor a con-

vertir, en primer lugar, y el tipo al que se desea convertir. Utilizando esta función la siguiente asignación sería completamente válida.

```
DedoAfectado = CType(2, Dedo)
Console.WriteLine(DedoAfectado.ToString())
```

La constante numérica entera 2 es convertida al tipo `Dedo`, obteniéndose en la variable `DedoAfectado` el valor `Corazón`.

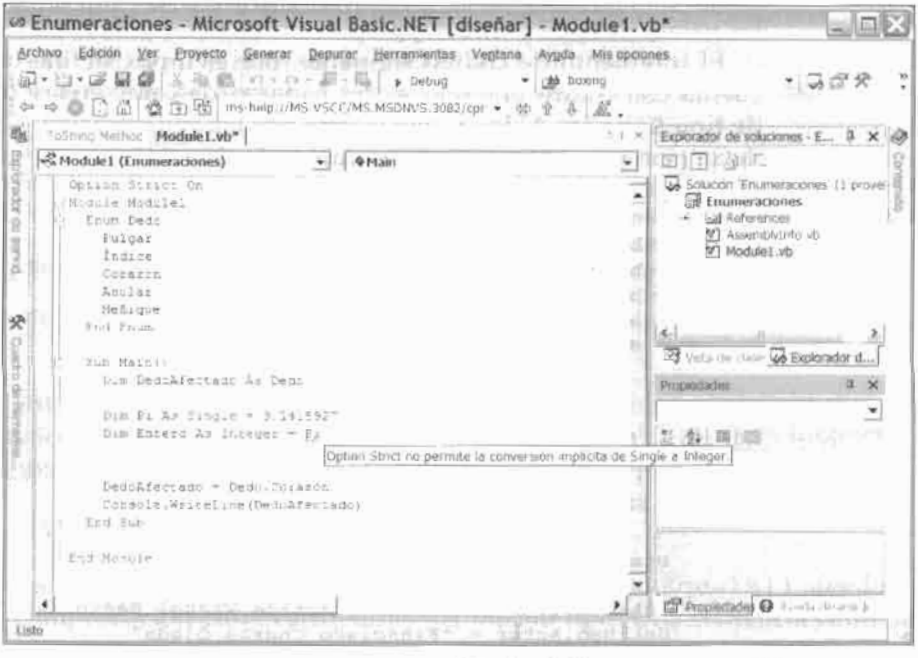


Figura 4.11. La asignación con conversión implícita ya no es posible

Nota

Debe tener en cuenta que ciertas conversiones implican la pérdida de parte de la información original. Es lo que ocurre, por ejemplo, cuando se asigna un número con parte decimal a una variable entera.

Estructuras

Las enumeraciones son tipos simples, al estilo de los enteros y caracteres. Una estructura, por el contrario, es un tipo complejo compuesto de uno o más miembros, cada uno de ellos pudiendo ser de cualquier tipo. Una estructura puede contar incluso con constructores y metodos, como las clases, sin, por ello, dejar de ser un tipo que mantiene un valor, no una referencia. Una estructura es el tipo adecuado cuando la complejidad de la información no es excesiva

y, por lo tanto, no justifica el diseño de una clase específica para gestionarla, creando objetos a partir de ella. Es el caso, por ejemplo, de estructuras como `Point`, `Size` o `Color`, definidas para mantener la información de un punto, unas dimensiones o un color. A pesar de las similitudes, hay que tener en cuenta que una estructura y una clase son entidades muy distintas. Las estructuras siempre están derivadas de `System.ValueType`, mientras que las clases pueden derivarse unas de otras. Las estructuras, por lo tanto, no cuentan con herencia. Las estructuras son en Visual Basic .NET el sustitutivo de los tipos de usuario de Visual Basic 6 y versiones previas. Básicamente, hay que sustituir las palabras `Type` y `End Type` por `Structure` y `End Structure`.

El fragmento de código siguiente, por ejemplo, define una estructura que cuenta con cuatro miembros: tres cadenas de caracteres y un valor numérico de tipo `Single`. Al igual que ocurre con las enumeraciones, partiendo de esta definición ya podemos crear variables de tipo `Libro`.

```
Structure Libro
  Public Titulo As String
  Public ISBN As String
  Public Autor As String
  Public Precio As Single
End Structure
```

Para acceder a los miembros con que contaría una variable de este tipo es necesario utilizar el operador `.` disponiéndolo entre el nombre de la variable y el del propio miembro. De esta forma, podríamos establecer u obtener los valores de la siguiente manera:

```
Dim UnLibro As Libro

UnLibro.Titulo = "Guía práctica Visual Basic .NET"
UnLibro.Autor = "Francisco Charte Ojeda"
UnLibro.ISBN = "84-415-1290-6"
UnLibro.Precio = 10.75
```

Para simplificar este código, en el que se hace referencia múltiples veces al identificador `UnLibro`, es posible utilizar la sentencia `With`. Con ella, y en el ámbito de un bloque delimitado por la propia palabra `With` y la terminación `End With`, es posible acceder a los miembros de la estructura utilizando tan sólo el operador `.` como se hace en el código mostrado a continuación.

```
With UnLibro
  .Titulo = "Guía práctica Visual Basic .NET"
  .Autor = "Francisco Charte Ojeda"
  .ISBN = "84-415-1290-6"
  .Precio = 10.75
End With
```

Asignación e igualdad

La variable `UnLibro` almacena directamente los valores asignados a los miembros de la estructura, siendo posible la asignación directa entre variables

del mismo tipo, como ocurre con el resto de los tipos que hemos conocido hasta ahora. No es posible, por el contrario, conversión alguna con variables de otros tipos, caso en el cual tendríamos que acceder de manera individual a los miembros que nos interesen para recuperar o asignar el valor.

En el código siguiente ve un caso hipotético en el que quiere asignarse el contenido de una variable `Libro` a otra del mismo tipo, algo totalmente correcto, tras lo cual deseamos asignar el valor de una cadena a la primera variable. Esto no es válido ni siquiera usando la función `CType()`. Como puede suponer, habría que escribir `UnLibro.Titulo = Titulo`.

```
Dim UnLibro, OtroLibro As Libro
Dim Titulo As String = "Programación con Visual Studio .NET"

With UnLibro
  .Titulo = "Guia practica Visual Basic .NET"
  .Autor = "Francisco Charte Ojeda"
  .ISBN = "84-415-1290-6"
  .Precio = 10.75
End With

OtroLibro = UnLibro
' UnLibro = CType(Titulo, Libro)
```

La comprobación de igualdad entre estructuras es otra operación que no podemos realizar como es habitual, utilizando el operador =. En el fragmento de código anterior, por ejemplo, no podríamos codificar un condicional así:

```
If UnLibro = OtroLibro Then
```

Todas las estructuras cuentan con un método, llamado `Equals()`, que facilita la comprobación entre estructuras del mismo tipo. Una sentencia como la siguiente mostraría por la consola `True` o `False`, dependiendo de que `UnLibro` y `OtroLibro` sean iguales o no.

```
Console.WriteLine(OtroLibro.Equals(UnLibro).ToString())
```

Miembros públicos y privados

Una diferencia de las estructuras de Visual Basic .NET, respecto a los tipos de usuario de versiones previas del lenguaje, es que los miembros no sólo cuentan con un identificador y un tipo sino que, además, deben precederse de un modificador que indique su visibilidad.

En la estructura `Libro` definida antes todos los miembros son `Public`, de tal manera que puede accederse a ellos desde cualquier punto donde se haya definido una variable `Libro`.

En lugar de `Public` podríamos haber usado la palabra `Dim`, obteniendo el mismo resultado. Las otras alternativas son `Private` y `Friend`. Con el primero los miembros serían privados, de tal manera que no podrían manipularse desde fuera de la propia estructura. Con el segundo tendríamos una combinación de `Public` y `Private`, puesto que los miembros de la estructura serían

públicos para el código del mismo ensamblado en el que se ha definido la estructura, permaneciendo como privados para otros ensamblados.

Posiblemente se pregunte qué sentido tendría declarar los miembros de una estructura como privados, de tal manera que no puedan manipularse directamente desde el exterior. La respuesta se encuentra en el punto siguiente.

Métodos, constructores y propiedades

A diferencia de los tipos de usuario de Visual Basic 6, las estructuras de Visual Basic .NET no sólo pueden contener miembros de datos sino que, además, pueden también contar con funciones y procedimientos actuando como métodos, que operan sobre los datos; constructores, que facilitan la inicialización, y propiedades, mediante los cuales controlar el acceso a esos datos.

En el capítulo dedicado a la programación orientada a objetos nos ocuparemos con mayor detalle, al estudiar las clases, de los constructores, la definición de propiedades y métodos. Por ahora nos conformaremos con ver cómo añadir un método y un constructor a la estructura `Libro`, dejándola como se muestra a continuación.

```
Structure Libro

  Public Sub New(ByVal T As String, ByVal I As String, _
                 ByVal A As String, ByVal P As Single)
    Titulo = T
    ISBN = I
    Autor = A
    Precio = P
  End Sub


  Public Sub Muestra()
    Console.WriteLine("{0}, {1}, {2}, {3}",
                  Titulo, ISBN, Autor, Precio)
  End Sub


  Private Titulo As String
  Private ISBN As String
  Private Autor As String
  Private Precio As Single
End Structure
```

Observe que ahora los miembros de datos son privados, no públicos, de tal manera que no pueden ser manipulados desde el exterior. Los dos procedimientos aparecen delante, aunque igualmente podríamos haberlos puesto detrás. El primero de ellos es el constructor debido a que hemos usado la palabra `New` como identificador. El segundo es un método corriente, al que podemos invocar cuando nos convenga.

Si declarásemos una variable de tipo `Libro` sin darle un valor inicial en ese momento, posteriormente no podríamos asignar un valor a sus miembros como si hacíamos en ejemplos previos. Usando el constructor, a través de la palabra

clave `New`, podríamos efectuar esa asignación, mientras que con el método `Muestra()` obtendríamos el contenido. Es lo que se hace en el ejemplo siguiente que, al ser ejecutado, genera el resultado de la figura 4.12. La primera variable está vacía, mientras que la segunda tiene los valores asignados en el momento de la creación.

```
Sub Main()
  Dim UnLibro As Libro
  Dim OtroLibro As Libro = New Libro( _
     "Introducción a la programación", _
     "84-415-1145-4", _
     "Francisco Charte Ojeda", _
     24.04)

  UnLibro.Muestra()
  OtroLibro.Muestra()
End Sub
```

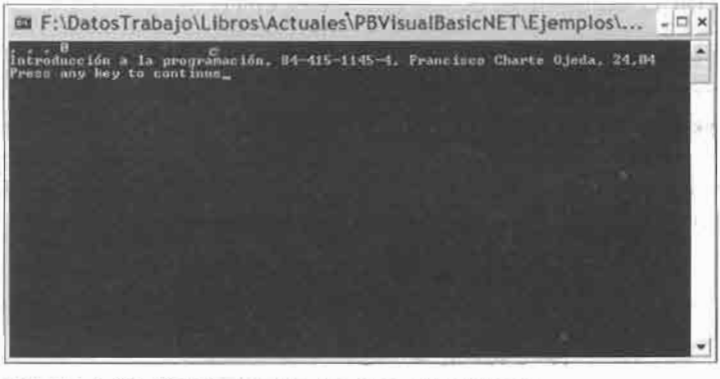


Figura 4.12. Contenido de las dos estructuras

Arrealos

Los tipos de datos usados hasta ahora, en ejemplos previos, tan sólo pueden contener un dato, ya sea simple: un número, un carácter o una cadena, o compuesto: una estructura. Cuando se tiene la necesidad de almacenar múltiples valores, del mismo tipo y con información relacionada, se utilizan los arreglos.

Un arreglo es una lista de variables del mismo tipo que comparten el nombre, distinguiéndose unas de otras por un índice. Los arreglos también son conocidos como *matrices*, aunque esta denominación es más apropiada para los arreglos de dos dimensiones. A diferencia de las estructuras, enumeraciones y demás tipos de datos conocidos hasta ahora, una variable de tipo arreglo no aloja el valor sino una referencia al bloque de memoria donde éste se encuentra.

Los arreglos, para ser más exactos cada uno de los elementos de un arreglo, pueden ser de cualquier tipo: enteros, cadenas, estructuras, referencias a objetos, etc. Incluso podemos tener arreglos de objetos creados a partir de una o

varias clases. A su vez, un arreglo también puede formar parte de un tipo complejo, como pueda ser una estructura.

Declaración de un arreglo

Como cualquier otra variable, un arreglo debe declararse antes de ser utilizado para almacenar valores. La declaración es, básicamente, igual que la de una variable corriente, debiendo especificarse el ámbito, identificador y tipo. Además, tras el identificador o nombre de la variable indicaremos, entre paréntesis, el mayor índice que tendrá.

A diferencia de Visual Basic 6, en Visual Basic .NET no es posible especificar los límites de un arreglo, su índice inferior y superior, ya que el índice inferior es siempre el 0. Basta, por tanto, con indicar cuál será el índice superior. El arreglo tendrá, por tanto, un elemento más del valor indicado como índice superior. En el ejemplo siguiente, por ejemplo, la variable `DiasMes` sería un arreglo con 12 elementos, desde el 0 hasta el 11.

```
Dim DiasMes(11) As Integer
```

No hay que olvidar que la variable `DiasMes` lo que contiene es una referencia al arreglo, y no el arreglo en sí. Esa referencia podemos usarla para acceder individualmente a cualquiera de los elementos, disponiendo su índice entre paréntesis:

```
DiasMes(0) = 31
DiasMes(3) = 30
```

Los índices, tanto al declarar la matriz como al acceder a los elementos, no tienen necesariamente que ser valores numéricos constantes. También podemos utilizar, por poner un ejemplo, las constantes de una enumeración, como se hace en el código siguiente:

```
Enum Mes
  Enero
  Febrero
  Marzo
  Abril
  Mayo
  Junio
  Julio
  Agosto
  Septiembre
  Octubre
  Noviembre
  Diciembre
End Enum

Sub Main()

  Dim DiasMes(Mes.Diciembre) As Integer
```

```
    DiasMes(Mes.Enero) = 31
    DiasMes(Mes.Abril) = 30
End Sub
```

El resultado es exactamente el mismo, pero el código queda mucho más claro utilizando esta notación en lugar de valores numéricos constantes.

Arreglos con varias dimensiones

El arreglo `DiasMes` declarado en el ejemplo anterior cuenta con una sola dimensión, de tal forma que puede representarse gráficamente como una lista de valores. El rango de esa dimensión es 0-11, valores que puede tomar su índice. Un arreglo puede tener varias dimensiones, hasta un número máximo de 60, cada una de ellas con un rango diferente.

El rango de cada dimensión puede contar con tantos elementos como valores posibles hay en el tipo `Long`.

Nota

En la práctica, con aplicaciones corrientes, no es habitual trabajar con arreglos de más de dos o tres dimensiones, mientras que el rango de cada una de ellas está físicamente limitado por la memoria disponible en el sistema, más que por los límites propios del lenguaje.

Suponga que desea contar con un arreglo en el que ir anotando una serie de datos estadísticos de cada una de las 24 horas de los 365 días del año. La declaración de dicho arreglo podría ser la siguiente:

```
Dim Medidas(364, 23) As Single
```

Mediante la variable `Medidas` podemos acceder a cualquiera de los 365 elementos de la primera dimensión, cada uno de los cuales cuenta con 24 elementos de la segunda dimensión. Al acceder a un elemento, por tanto, es necesario facilitar dos índices: el número del día y el de la hora a la que corresponde la medida a almacenar o recuperar. El número total de elementos en el arreglo será de 365 por 24: 8760 elementos.

Rangos dinámicos

Aunque en los fragmentos de código anterior, al declarar un arreglo, siempre hemos indicado el índice superior del rango de cada dimensión, nada nos impide declarar arreglos sin un rango inicial. Si es necesario, sin embargo, indicar el número de dimensiones con que contará. Por ejemplo, si no supiéramos de antemano cuántos días vamos a tomar medidas, en el supuesto del

punto anterior, ni cuántas horas de cada día, podríamos declarar el arreglo `Medidas` así:

```
Dim Medidas(,) As Single
```

Mientras no establezcamos el rango de las dimensiones de este arreglo, la variable `Medidas` tiene una referencia nula. Cualquier acceso usando esta referencia causaría una excepción. Puede utilizar la función `TypeName()` para obtener el tipo de cualquier variable, o bien usar una expresión `TypeOf Is` para comprobar su tipo. Puede comprobarlo con el código siguiente:

```
Dim Medidas(,) As Single

Console.WriteLine(TypeName(Medidas))
' Esta expresión devuelve False
Console.WriteLine(TypeOf (Medidas) Is Array)
```

El resultado de ejecutar este código es el de la figura 4.13. Puede ver que el valor devuelto por `TypeName()` es `Nothing`, indicando que `Medidas` no tiene referencia alguna en ese momento. La comprobación de si el tipo de la variable es `Array` da como resultado el valor `False`.

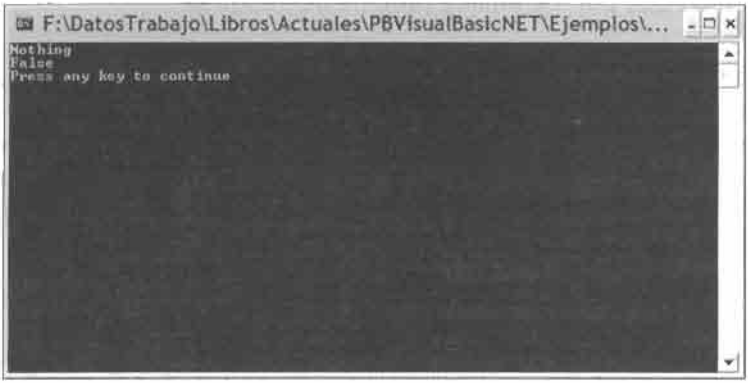


Figura 4.13. La variable `Medidas` no tiene una referencia válida

Nota

El tipo `Array` es una clase de la cual están derivados implícitamente todos los arreglos que declaremos. A continuación conocerá algunos de los métodos de esa clase, heredados automáticamente por los arreglos.

Para establecer el número de elementos de cada dimensión de la variable `Medidas` usaremos la instrucción `Redim`. Esta debe ir seguida del identificador

del arreglo, `Medidas` en este caso, y, entre paréntesis, los valores para establecer el índice superior de cada dimensión. Esos valores pueden ser, en lugar de constantes, datos introducidos por el usuario o recuperados de una base de datos. En el siguiente fragmento de código, por ejemplo, se pide por teclado el número de días y horas durante las que van a tomarse medidas, usando esos valores para establecer el número de elementos del arreglo.

```
Dim Dias, Horas As Short

Console.Write("Introduzca el número de días: ")
Dias = Conversion.Val(Console.ReadLine())
Console.Write("Introduzca las horas por día: ")
Horas = Conversion.Val(Console.ReadLine())


ReDim Medidas(Dias - 1, Horas - 1)
```

El método `ReadLine()` de la clase `Console` devuelve una cadena de caracteres con el valor introducido por el usuario. Dado que nosotros necesitamos el valor numérico, no la secuencia de caracteres que lo componen, usamos la función de conversión `Val()` para realizar esa transformación. Finalmente nos servimos de los datos obtenidos para establecer el número de elementos del arreglo. Fíjese en que restamos uno tanto al número de días como de horas, ya que lo que hay que facilitar a `ReDim` no es el número de elementos sino el índice más alto de cada dimensión. Si tras establecer las dimensiones del arreglo, con la sentencia `ReDim`, comprobamos el tipo de la variable `Medidas` veremos que ya no es `Nothing`, sino `Single(,)`, y que la comprobación `TypeOf(Medidas) Is Array` devuelve `True` en lugar de `False`.

La instrucción `ReDim` no es útil sólo para establecer el número de elementos inicial de un arreglo, pudiendo utilizarse también para modificar dicho número cuando interese. El rango de elementos de cada dimensión no es algo estático, sino dinámico, de ahí que podamos alterarlo cuando nos convenga. Si el cambio provoca una reducción del número de elementos, está claro que provocará una pérdida de información puesto que los elementos sobrantes serán descartados. Si, por el contrario, lo que se hace es ampliar el número de elementos, los ya existentes pueden conservar el valor que tenían simplemente añadiendo tras `ReDim` la palabra `Preserve`.

Información sobre un arreglo

Como se ha indicado antes, todos los arreglos son objetos derivados de la clase `Array`. Dicha clase cuenta con un importante número de miembros, propiedades y métodos principalmente, que nos permiten tanto operar sobre el arreglo como obtener información acerca de él. En este momento nos interesa lo segundo, recuperar algunos datos relativos al arreglo sobre el que estamos trabajando. Esto es especialmente interesante si estamos usando un arreglo dinámico que va creciendo o decreciendo según las necesidades del programa.

Algunos datos son realmente útiles, como el contenido en la propiedad `Rank`, que nos indica el número de dimensiones, o los devueltos por los métodos `GetLength()`, `GetLowerBound()` y `GetUpperBound()` que son el número de elementos, el índice inferior y superior, respectivamente, de la dimensión cuyo número se entrega como parámetro. Al igual que los índices, las dimensiones se numeran desde 0 hasta el valor contenido en `Rank` menos 1.

Otras propiedades tan sólo son útiles en caso de que se derive una nueva clase tomando `Array` como base. Es el caso de `IsFixedSize` o `IsReadOnly`, ambas de tipo `Boolean`. La primera indica si el arreglo tiene un tamaño fijo y la segunda si es de sólo lectura, devolviendo siempre los valores `True` y `False`, respectivamente.

Conociendo estos miembros de `Array`, y sólo a título informativo, podríamos codificar un método como el siguiente:

```
Sub InformeArreglo(ByVal Arreglo As Array)
  Dim Dimension As Byte


  If TypeOf (Arreglo) Is Array Then
    Console.WriteLine("____________________________________")
    Console.WriteLine("Tipo del arreglo: {0}", Arreglo.GetType())
    Console.WriteLine("IsFixedSize = {0}", Arreglo.IsFixedSize)
    Console.WriteLine("IsReadOnly = {0}", Arreglo.IsReadOnly)
    Console.WriteLine("{0} dimensiones", Arreglo.Rank)
    For Dimension = 0 To Arreglo.Rank - 1
      Console.WriteLine("Dimensión {0} tiene {1} elementos", _
            Dimension, Arreglo.GetLength(Dimension))
      Console.WriteLine("Índice inferior {0}, superior {1}", _
            Arreglo.GetLowerBound(Dimension), _
            Arreglo.GetUpperBound(Dimension))
    Next
  End If
End Sub
```

Como puede ver, primero comprobamos que el parámetro recibido es de tipo `Array`, ya que podría ser nulo, tras lo cual procedemos a mostrar diversos datos sobre él. El método `GetType()` facilita el tipo de una variable. En este caso simplemente lo usamos para mostrar el nombre de dicho tipo. Podríamos llamar varias veces a este método, desde `Main()`, facilitando distintos arreglos. El resultado sería similar al de la figura 4.14.

Asignación y copia

Cuando asignamos una variable que contiene un valor, como puede ser un entero o una estructura, a otra variable del mismo tipo, lo que hacemos es copiar el valor de una variable a otra. Tenemos, por tanto, dos variables con el mismo valor, pero independientes entre sí. La situación es totalmente distinta al tratar con arreglos ya que, como se dijo anteriormente, son objetos derivados de `Array` y, por tanto, lo que tenemos en las variables son referencias.

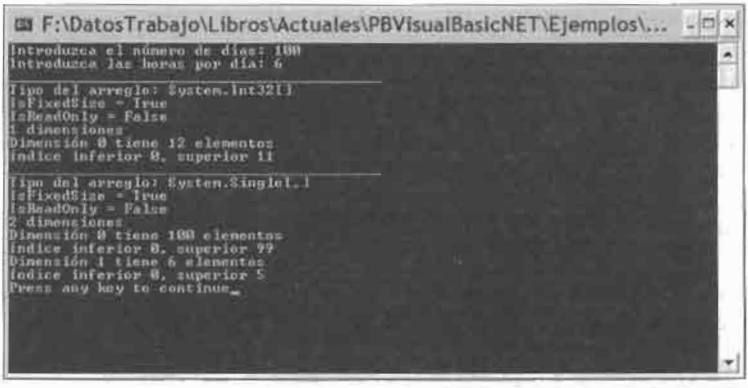


Figura 4.14. Información relativa a dos arreglos

Al asignar una variable de tipo arreglo a otra, algo que es posible siempre que tengan el mismo número de dimensiones y tipo, lo que se hace es duplicar la referencia. Es decir, contaremos con un solo arreglo pero con dos referencias a él en dos variables distintas. Podríamos, por ejemplo, modificar los elementos mediante una variable y leerlos con la otra ya que, como se ha dicho, en realidad existe un solo arreglo, no dos.

Partiendo de los ejemplos anteriores, en los que teníamos diversos arreglos entre ellos uno con dos dimensiones, podríamos usar el código siguiente para efectuar una comprobación simple:

```
Dim Medidas(,) As Single
Dim CopiaMedidas(,) As Single

ReDim Medidas(100, 6)

CopiaMedidas = Medidas

CopiaMedidas(0, 0) = 10.7
Console.WriteLine(Medidas(0, 0))
```

En principio `Medidas` es un arreglo vacío, sin embargo al llamar al método `WriteLine()` para mostrar el elemento `(0,0)` obtenemos el dato asignado mediante `CopiaMedidas`, otra referencia al mismo arreglo.

Si lo que deseamos es obtener una copia del arreglo, obteniendo una referencia a ese nuevo arreglo, debemos usar el método `Clone()`. Es lo que se hace en la sentencia siguiente, asumiendo que `Medidas` y `CopiaMedidas` ya están declaradas:

```
CopiaMedidas = Medidas.Clone()
```

El método `Clone()` crea un nuevo arreglo y copia en él todos los elementos del original. Si dichos elementos contienen valores, como es el caso de `Medidas`, lo que se obtiene es una copia del valor. Si, por el contrario, los elementos

almacenan referencias, `Clone()` no copia los objetos a los que apuntan dichas referencias, es decir, se limita a copiar el arreglo.

Una alternativa a `Clone()` es el método `Copy()` de la clase `Array`. Este permite copiar parte de los elementos de un arreglo a otro arreglo, si es necesario incluso efectuando una conversión entre tipos compatibles.

Otras operaciones con arreglos

Como se ha repetido en varias ocasiones, los arreglos son objetos, no simples valores, de clases derivadas de `Array`. Este hecho abre un abanico de posibilidades impresionante, especialmente si lo comparamos con las posibilidades existentes en Visual Basic 6.

La clase `Array` está derivada de `Object`, por lo que ya dispone de métodos existentes en todos los objetos de Visual Basic .NET, tales como `GetType()` o `ToString()`. Además, `Array` implementa varias interfaces, como `ICloneable` o `IEnumerable`, que ofrecen métodos para simplificar la copia o enumeración del arreglo.

Gracias a la existencia de métodos como `Sort()`, `Reverse()`, `IndexOf()` y `BinarySearch()` es posible ordenar los elementos de un arreglo, invertir su orden, obtener el índice del elemento que contiene un cierto valor o efectuar una búsqueda binaria en un arreglo ordenado, especialmente útil cuando el número de elementos es muy grande. Todas estas operaciones, y algunas más, se describen en la referencia en línea de la clase `Array` que puede encontrar en la ayuda de Visual Basic .NET.

Si quiere realizar una prueba simple, ordenando y enumerando los elementos de un arreglo, puede usar el código siguiente:

```
Dim Meses() As String = {_
  "Enero", "Febrero", "Marzo", "Abril", _
    "Mayo", "Junio", "Julio", "Agosto", _
    "Septiembre", "Octubre", "Noviembre", _
    "Diciembre"}
Dim Valor As String

Console.WriteLine("Antes de ordenar")
Console.WriteLine("----------------")

For Each Valor In Meses
  Console.WriteLine(Valor)
Next

Array.Sort(Meses)

Console.WriteLine("Despues de ordenar")
Console.WriteLine("----------------")

For Each Valor In Meses
  Console.WriteLine(Valor)
Next
```

Figura 4.15. La lista de meses antes y después de ser ordenada

Nota

Cuando no se indica explícitamente el índice de un arreglo, al declararlo, pero se facilita una lista de valores iniciales, entre llaves, el propio compilador se encarga de establecer el número de elementos.

La construcción `For Each/Next`, usada en este ejemplo para enumerar el contenido del arreglo `Meses`, utiliza una de las interfaces de `Array` para recorrer todos los elementos del arreglo sin necesidad de conocer sus índices inferior y superior.

Trabajo con caracteres

Visual Basic .NET dispone de dos tipos de datos que facilitan la manipulación de caracteres: `Char` y `String`. El primero de ellos es un tipo simple, al estilo de `Integer` o `Byte`, mientras que el segundo es una clase, como `Array`. A pesar de su naturaleza de tipo intrínseco, como ya sabe puede usar el empaquetamiento para introducir un carácter en el tipo `System.Char`, obteniendo un objeto con el beneficio que ello conlleva.

Una variable de tipo `Char` puede contener cualquier carácter del conjunto Unicode. En la plataforma .NET no existen caracteres ASCII, por lo que no es necesario estar efectuando conversiones entre los dos conjuntos. El conjunto Unicode facilita la representación de miles de caracteres de todos los alfabetos, en contraposición a ASCII que, prácticamente, estaba pensado para la lengua anglosajona.

La mayoría de los métodos con que cuenta el tipo `Char` están destinados a facilitar la clasificación de los caracteres. Todos ellos tienen un nombre que comienza por `Is` y devuelven un valor de tipo `Boolean`. De esta forma podemos

saber si el carácter es un dígito decimal: `IsDigit()`, una letra: `IsLetter()`, un carácter de puntuación: `IsPunctuation()`, etc. También podemos convertir el carácter a mayúscula, minúscula o bien una cadena mediante los métodos `ToUpper()`, `ToLower()` y `ToString()`, respectivamente.

Cadenas de caracteres

A diferencia de los caracteres individuales, las cadenas de caracteres son tratadas en Visual Basic .NET como objetos. Una variable de tipo `String` almacena una referencia al lugar en el que se encuentra la cadena de caracteres. Cada vez que se modifique dicho valor se asigna un nuevo bloque de memoria, se almacena en él la nueva cadena y la anterior se libera, almacenándose en la variable `String` la nueva referencia.

Los métodos de la clase `String` nos permiten conocer la longitud de la cadena de caracteres, obtener una parte de ella, reemplazarla, copiarla, acceder a los caracteres individuales, eliminar espacios sobrantes, ajustar a izquierda o derecha hasta una longitud indicada, etc. No hay mas que echar un vistazo a la lista de miembros de la clase `String` para saber las posibilidades que tenemos a nuestro alcance. La mayoría de los miembros son de uso simple.

Al igual que en Visual Basic, podemos usar el operador `&` para unir dos cadenas obteniendo la concatenación. Alternativamente puede utilizar el método `Concat()` para obtener exactamente el mismo resultado.

Manipulación intensiva de cadenas

El hecho de que cada vez que se manipula una cadena, por ejemplo concatenando otra o extrayendo una porción de la existente, se cree un nuevo objeto en memoria tiene importantes consecuencias en aquellos casos en los que se precisa un trabajo intensivo de manipulación de cadenas. El tiempo consumido cada vez que se manipula la cadena, especialmente si ésta es muy grande, puede ser considerable al repetirse miles de veces. Puede comprobarlo con el código siguiente:

```
Dim Caracteres As String

Dim Contador As Integer

Dim Ahora As DateTime

Console.Write("Usando el operador & de String: ")
Ahora = Now

For Contador = 1 To 32000
  Caracteres = Caracteres & " - "
Next

Console.WriteLine(Now.Subtract(Ahora))
```

Básicamente, añadimos tres caracteres a una cadena 32000 veces. Lo que ocupa la mayor parte del tiempo no es el bucle en sí, sino la tarea de copiar la cadena que cada vez es más grande. Para controlar el tiempo transcurrido tomamos una referencia al inicio del bucle, mediante la propiedad `Now`, y hacemos lo mismo al final del bucle obteniendo la diferencia. Ésta vendrá expresada en forma de horas, minutos, segundos y pulsos de reloj.

Podríamos intentar utilizar el método `Concat()`, anteriormente citado, en lugar del operador `&`. Veríamos que el tiempo empleado en la operación es prácticamente el mismo, en el caso de mi equipo casi treinta segundos en ambos casos.

Cuando las cadenas de caracteres usadas en un programa no sean tan sólo valores más o menos estáticos, usados en ciertos contextos donde son necesarios, sino que vaya a efectuarse una manipulación relativamente intensiva, en lugar del tipo `String` nos interesará utilizar el nuevo tipo `StringBuilder`.

`StringBuilder` es una clase que se caracteriza por preparar un bloque de memoria en el que se almacenará la cadena que va a manipularse, no reasignándose este espacio, ni copiándose la cadena a cada manipulación, hasta que dicha capacidad haya llegado a su límite. Podemos dejar que la propia clase `StringBuilder` gestione la capacidad del bloque de memoria, que tendrá un espacio inicial de 16 caracteres y que irá creciendo cuando sea necesario, o bien indicar de manera explícita dicha capacidad en el momento de crear el objeto.

La clase `StringBuilder` cuenta con un conjunto de métodos similar al de `String`, facilitando la concatenación de cadenas, eliminación y sustitución de partes de la cadena. Estas operaciones se efectúan siempre sobre la cadena almacenada en el bloque de memoria inicialmente asignado, sin ninguna sobrecarga adicional. Puede ver la diferencia considerable de rendimiento usando un código como el mostrado a continuación:

```
Sub Main()
  Dim Caracteres As String
  Dim Contador As Integer
  Dim Ahora As DateTime

  Console.Write("Usando el operador & de String: ")
  Ahora = Now
  For Contador = 1 To 32000
    Caracteres = Caracteres & " - "
  Next
  Console.WriteLine(Now.Subtract(Ahora))

  Caracteres = Nothing

  Console.Write("Usando el método Concat() de String: ")
  Ahora = Now
  For Contador = 1 To 32000
```

```
    Caracteres = String.Concat(Caracteres, " - ")
  Next
  Console.WriteLine(Now.Subtract(Ahora))

  Console.Write("Usando el método Append() de StringBuilder: ")
  Dim Caracteres2 As New Text.StringBuilder()
  Ahora = Now
  For Contador = 1 To 96000
    Caracteres2.Append(" - ")
  Next
  Console.WriteLine(Now.Subtract(Ahora))
End Sub
```

Como se observa en la figura 4.16, a pesar de que en el `StringBuilder` hemos añadido tres veces más caracteres, y el propio bucle consume mucho más tiempo al repetirse el triple de veces, apenas consume en total media décima de segundo. Los otros dos métodos, usando una variable `String`, emplean, comparativamente hablando, muchísimo más tiempo.

Figura 4.16. Comparativa de tiempos entre los tres métodos usados

Constantes

En la mayoría de los ejemplos propuestos a lo largo de este capítulo se han usado valores constantes, principalmente numéricas y cadenas de caracteres, confiando la deducción de su tipo al lenguaje en base al propio valor. Esa deducción, no obstante, en ocasiones no resulta tan obvia. Al encontrar la constante `"P"`, por ejemplo, Visual Basic .NET la trata como de tipo `String`, pero pudiera ser que a nosotros nos interesase que la tomara como un carácter, tipo `Char`. Algo similar puede ocurrir con un número cualquiera. El valor `3.1416`, por ejemplo, sería interpretado como `Double` esperando nosotros que lo fuera como `Single`. Aunque podríamos servirnos de funciones de conversión como `CChar()`, `CDbl()` y similares, lo cierto es que, en el caso de las constantes literales, resulta más fácil utilizar los sufijos enumerados en la tabla 4.3.

Tabla 4.3. Sufijos indicativos de tipo para constantes literales

Sufijo	Tipo	Ejemplo
C	Char	"P"C
I	Integer	34I
S	Short	34S
L	Long	34L
R	Double	3.1416D
F	Single	3.1416F
D	Decimal	34D

Nota

Se conoce como constantes literales a los valores introducidos como tales en el código del programa, sin estar representados por identificador alguno. Son literales `34`, `True` o `"P"`.

Una alternativa al uso de literales y sufijos, especialmente recomendable porque contribuye a hacer más claro el código, consiste en representar las constantes mediante identificadores, como en el caso de las variables. La sintaxis para declarar una constante es, de hecho, prácticamente idéntica a la de una variable, tan sólo hay que añadir la palabra `Const` delante del identificador. Las constantes pueden aparecer en los mismos puntos donde puede declararse una variable, así como contar con los mismos modificadores de visibilidad.

El uso de constantes no literales hace que el código quede más claro. En un ejemplo anterior usábamos dos constantes, 365 y 24, para establecer el número de elementos de un arreglo. También podríamos hacerlo así:

```
Const Dias As Short = 365
Const Horas As Byte = 24

Dim Medidas(Dias - 1, Horas - 1) As Single
```

No sólo el código queda más claro, sino que una hipotética modificación de los valores 365 y 24 resultaría mucho más simple: bastaría con cambiar la declaración de las constantes. Si hubiésemos utilizado las literales, nos veríamos forzados a buscar su aparición por todo el código.

Puntos clave

- Todos los tipos de datos de Visual Basic .NET tienen al tipo `Object` con raíz común, dando lugar a un sistema de tipos totalmente unificado en el que todos los valores pueden ser tratados como objetos.

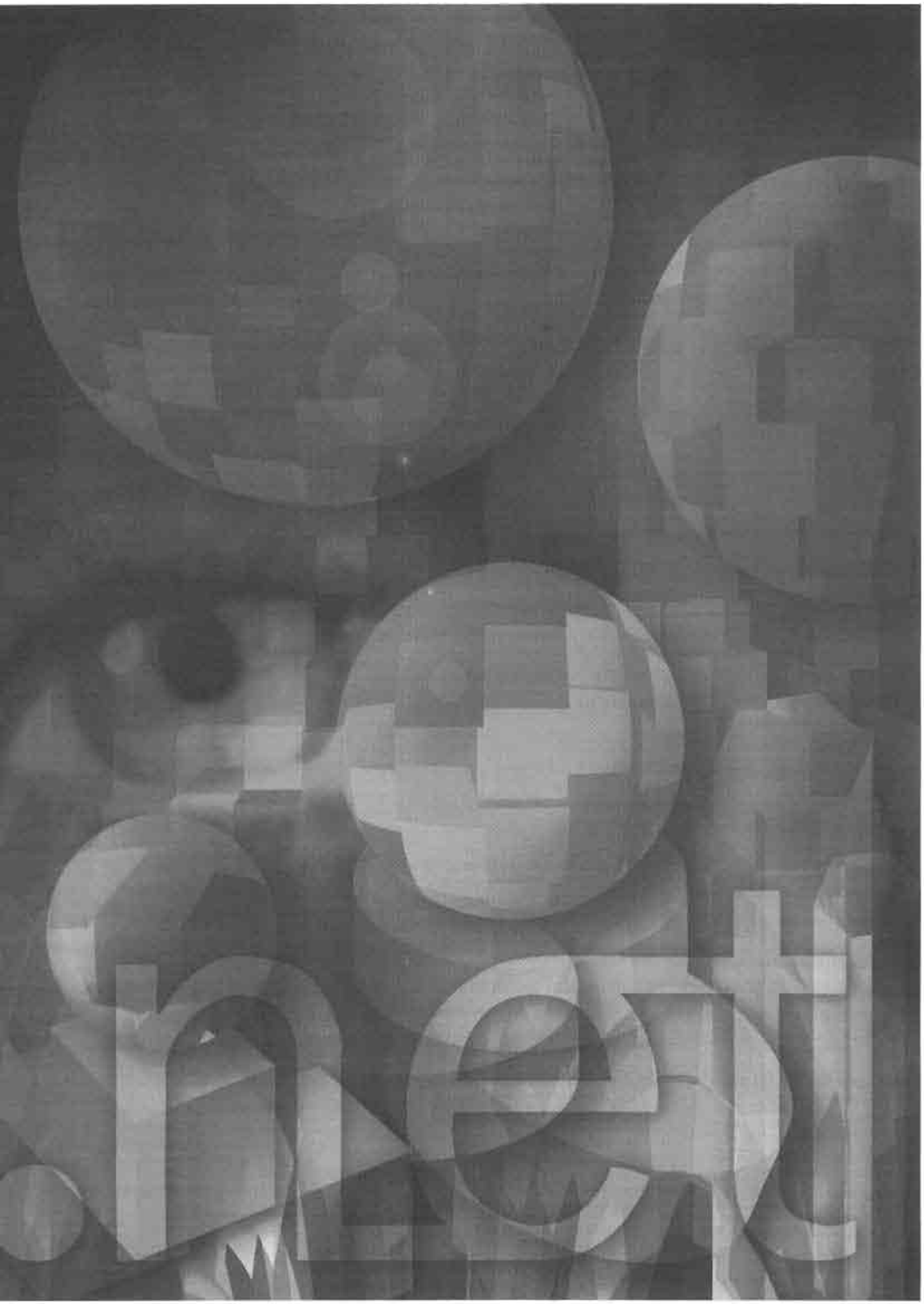
- Los tipos de datos pueden clasificarse en dos grandes ramas: los derivados de `ValueType`, que almacenan directamente el valor, y todos los demás, con una referencia a un bloque de memoria donde se encuentra ese valor.
- Visual Basic .NET sólo contempla como nativos los tipos de datos de la plataforma .NET que forman parte de la especificación CLS.
- Las variables, constantes y enumeraciones pueden tener básicamente tres ámbitos: local a un método o bloque de código, local a un módulo o clase o global a todo un ensamblado.
- Las estructuras tan sólo pueden definirse al nivel de módulo o clase, pero no en el interior de una función.
- En Visual Basic .NET es posible declarar múltiples variables de un mismo tipo especificando éste una sola vez, al final.
- En el mismo momento de la declaración es posible la asignación de un valor inicial.
- Las enumeraciones posibilitan la creación de nuevos tipos de datos simples, basados en el uso de un subrango de enteros.
- Mediante las estructuras podemos crear tipos complejos, con múltiples miembros de cualquier tipo, además de métodos y constructores.
- Los arreglos son listas o matrices de elementos que, compartiendo un mismo nombre y tipo, se distinguen entre sí por uno o varios índices.
- En Visual Basic .NET el índice inferior de un arreglo siempre es el cero, mientras que el superior se especifica en el momento de la declaración o con la instrucción `ReDim`.
- El número de dimensiones de un arreglo es fijo desde el momento en que se declara, mientras que el número de elementos de cada dimensión es un factor dinámico.
- Todos los arreglos son objetos derivados de `Array`, por lo que podemos usar los métodos de dicha clase con el fin de operar sobre el arreglo u obtener información sobre él.
- Para trabajar con cadenas de caracteres Visual Basic .NET dispone de dos tipos distintos: `String` y `StringBuilder`. El primero es el usado habitualmente, mientras que el segundo obtiene un mejor rendimiento a la hora de una manipulación intensiva de las cadenas.
- Gracias al uso de constantes es posible clarificar el código y facilitar su mantenimiento.

Resumen

Al finalizar este capítulo tenemos la información suficiente como para utilizar cualquiera de los tipos de datos básicos de Visual Basic .NET, usándolos

para declarar variables, definir constantes e introducir literales en el código. También hemos aprendido a crear nuestros propios tipos de datos, mediante enumeraciones y estructuras. Finalmente, hemos visto cómo crear y usar arreglos, así como el uso básico de las cadenas de caracteres.

Muchos de los elementos vistos pueden combinarse entre sí, dando lugar a construcciones más complejas. En una estructura, por ejemplo, podemos contar con uno o más miembros que sean arreglos o estructuras. Los elementos de un arreglo no tienen por qué ser enteros o cadenas de caracteres, también pueden ser estructuras, objetos e, incluso, otros arreglos. Son casos puntuales con los que irá encontrándose a medida que vaya desarrollando proyectos con Visual Basic .NET.


.net

5 Operadores y expresiones

En el capítulo previo hemos conocido los tipos de datos intrínsecos de Visual Basic .NET, así como la manera de usarlos para construir nuestros propios tipos, declarar variables y constantes o usarlos como elementos de arreglos. Los valores, independientemente de dónde se encuentren almacenados, se utilizan en su mayor parte como operandos de expresiones con el fin de producir resultados.

Una expresión es una combinación de operandos, los valores sobre los que va a actuarse, y operadores, que indican las operaciones que van a efectuarse, obteniendo un solo valor final. Las expresiones pueden ser aritméticas, relacionales y lógicas, combinándose muchas veces entre sí formando expresiones complejas.

Nuestro objetivo, en este capítulo, es conocer la mayoría de los operadores con que cuenta Visual Basic .NET, algunos de los cuales ya hemos conocido en ejemplos del capítulo anterior. Es el caso del operador de asignación, =, y concatenación de cadenas, &. Con esto daremos un paso más en el conocimiento de los elementos del lenguaje, especialmente de aquellos que resultan fundamentales como es el caso de los operadores.

Asignación de valores

Seguramente la expresión más simple que podemos encontrar es la de asignación. En ella interviene un operador y dos operandos: el valor a asignar, a la

derecha del operador =, y el receptor, a su izquierda. El receptor puede ser una variable, un elemento de un arreglo, un miembro de una estructura o una clase o una propiedad. En general, cualquier identificador que sea capaz de contener un valor del tipo que pretende asignarse.

El valor puede ser una constante, literal o no, otra variable del mismo tipo o bien una expresión que devuelva un valor válido. En la mayoría de los ejemplos del capítulo anterior hemos efectuado diversas asignaciones utilizando el operador =.

Una novedad de Visual Basic .NET es la existencia de operadores abreviados para asignación de expresiones en las que el primer operando es el propio receptor del valor resultante. Imagine que quiere incrementar el valor contenido en una variable. En Visual Basic 6 usaría una expresión como la siguiente:

```
Valor = Valor + 1
```

En Visual Basic .NET esta expresión puede simplificarse utilizando el operador +=, tal y como se hace a continuación.

```
Valor += 1
```

Como puede suponer, el resultado es exactamente el mismo. De hecho, el código MSIL generado por ambas sentencias es idéntico. Además del operador +=, como versión abreviada del operador aritmético +, también tenemos a nuestra disposición los operadores -=, *=, /=, \=, ^= y &=, equivalentes a -, *, /, \, ^ y & pero asumiendo que el primer operando de la expresión será la misma variable que va a recibir el resultado.

Operadores aritméticos

La finalidad de esta familia de operadores es permitirnos realizar el conjunto de operaciones matemáticas más básico, como son la suma, resta, multiplicación y división. Cualquiera de estos operadores siempre toma dos operandos y genera un resultado. En la tabla 5.1 podemos ver un resumen de estos operadores, junto con la operación que realizan. En esta tabla **N1** representa al primer parámetro, el que se sitúa a la izquierda del operador, y **N2** al segundo, a la derecha del operador.

Tabla 5.1. Operadores aritméticos

Operador	Operación aritmética que efectúa
+	Suma de N1 y N2
-	Halla la diferencia entre N1 y N2
*	Multiplica N1 por N2
/	Divide N1 entre N2

Operador	Operación aritmética que efectúa
\	Divide N1 entre N2 sin hallar decimales (división entera)
^	Eleva N1 al exponente indicado por N2
Mod	Resto de la operación N1 \ N2

Estos operadores, como es obvio, trabajan sobre cualquiera de los tipos numéricos que conocemos, enteros o coma flotante, independientemente de su precisión. Sin embargo, cuando en una misma expresión utilicemos operandos de distinto tipo numérico Visual Basic realizará las conversiones necesarias para guardar la mayor precisión posible. Trabajando con números los primeros cuatro operadores no requieren ningún comentario adicional, se limitan a realizar las cuatro operaciones aritméticas indicadas. El operador \ realiza la división entera, es decir, la única diferencia con el operador / es que este último puede generar una parte decimal en el cociente de la división, mientras que el primero no.

Precisamente para hallar el resto de esa división entera podemos utilizar el operador `Mod`, que toma exactamente los mismos parámetros y realiza la misma operación que \, la división entera, con la única diferencia de que no nos devuelve el cociente obtenido, sino el resto.

Por último tenemos el operador ^, que nos permitirá realizar potenciaciones, actuando como base el operando que se sitúe a la izquierda y como exponente el que se encuentre a la derecha.

Por medio del operador + podemos trabajar, además, con cadenas de caracteres, tipo `String`, siendo en este caso el resultado la concatenación de ambas. Obtendremos, por tanto, una nueva cadena formada por la primera más la segunda. En Visual Basic existe un operador, &, que tiene precisamente la finalidad de concatenar dos datos formando una cadena, siendo preferente el uso de este operador sobre +.

En el caso de que en una misma expresión se utilicen distintos operadores aritméticos y no se alteren las prioridades, por medio de paréntesis, siempre se evaluarán primero las potenciaciones, después los operadores /, *, \ y `Mod` de izquierda a derecha, y por último + y - de izquierda a derecha.

Operaciones con variables `Object`

Cuando realicemos operaciones aritméticas con variables del tipo `Object` el resultado vendrá dado por el tipo de dato que en ese momento contenga la variable, que puede ser numérico, de cadena, fecha u otro tipo, y el segundo operando, alterando el tipo del receptor del resultado cuando sea necesario. De esta forma, una simple suma puede cambiar el tipo de dato de `Integer` a `Double` o a `String`, según los casos. Observe el código siguiente:

```
Dim Valor As Object = 5

Console.WriteLine("{0} -> {1}", Valor, Valor.GetType.ToString())
```

```
Valor = Valor + 1
Console.WriteLine("{0} -> {1}", Valor, Valor.GetType.ToString())
Valor += 1.5
Console.WriteLine("{0} -> {1}", Valor, Valor.GetType.ToString())
Valor &= " pesetas"
Console.WriteLine("{0} -> {1}", Valor, Valor.GetType.ToString())
```

En el contamos con una variable de tipo `Object` que, inicialmente, contiene el valor numérico 5. Recuperamos el tipo, de la propiedad `GetType`, y lo mostramos como una cadena de caracteres, con `ToString()`. Acto seguido sumamos un valor numérico entero y mostramos de nuevo el tipo, que no se habrá visto alterado. Al sumar un número con parte decimal, sin embargo, el tipo de la variable cambia, y lo mismo ocurre al concatenar una cadena. Es algo que puede ver en la figura 5.1, que es el resultado obtenido al ejecutar ese código.

Lógicamente, utilizando una variable de un tipo específico, como `Integer`, esas conversiones automáticas no podrían tener lugar. También podemos evitarla en el tipo `Object` simplemente activando la comprobación estricta de tipos de Visual Basic .NET citada en un capítulo previo, bien disponiendo la sentencia `Option Strict On` al inicio del módulo o modificando la opción equivalente en la ventana de propiedades del proyecto.

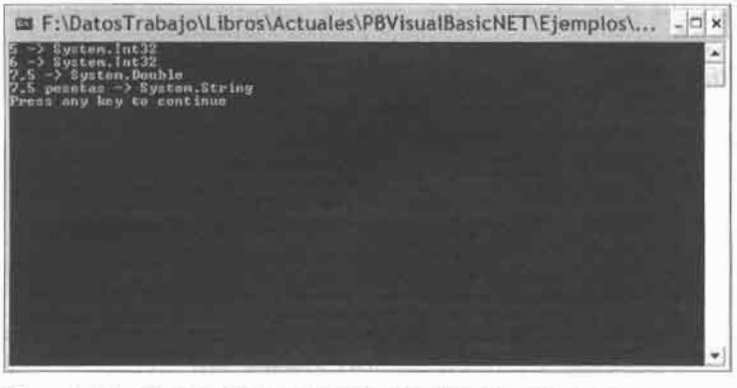


Figura 5.1. El tipo de la variable cambia dinámicamente

Operadores relacionales

Este conjunto de operadores nos permitirán evaluar expresiones y obtener dos resultados posibles: `True`, si la expresión es cierta o devuelve un valor distinto de cero, o `False`, si la expresión es falsa o devuelve cero. También se les conoce como operadores de comparación. Se utilizan principalmente en estructuras de decisión, en las que dependiendo del valor obtenido se toma un camino u otro, y de control de repetición.

En la tabla 5.2 se enumera cada uno de los operadores de comparación, junto con una pequeña descripción en la que **N1** representa al primer operando y **N2** al segundo.

Tabla 5.2. Operadores relacionales o de comparación

Operador	Devuelve `True` si	Devuelve `False` si
=	N1 y N2 son iguales	N1 y N2 son distintos
<>	N1 y N2 son distintos	N1 y N2 son iguales
<	N1 es menor que N2	N1 es igual o mayor que N2
<=	N1 es menor o igual a N2	N1 es mayor que N2
>	N1 es mayor que N2	N1 es igual o menor que N2
>=	N1 es mayor o igual a N2	N1 es menor que N2

Por medio de estos operadores podemos comparar números, cadenas y fechas. En el caso de los números será su valor el que intervenga directamente en la relación. Si comparamos dos cadenas en realidad se compararán los códigos de cada uno de los caracteres que la componen, de tal forma que `"Mañana"` no será igual a `"mañana"`, ya que el código de la `'M'` no es el mismo que `'m'`.

Al trabajar con dos fechas se utilizará la representación interna de las variables tipo `Date` para saber si una fecha es igual, anterior o posterior a otra. Hay que tener en cuenta que una variable del tipo `Date` además de la fecha también puede contener la hora, por lo que al realizar una comparación puede que no obtengamos el resultado esperado. Por ejemplo, dos variables `Date` pueden contener la fecha del día de hoy, pero si una tiene una hora y otra no, o tiene una hora distinta, las variables no cumplirán la relación de igualdad.

Nota

En la comparación de cadenas interviene una opción de Visual Basic .NET, `Option Compare`, que por defecto tiene el valor `Binary`. Introduciendo la sentencia `Option Compare Text` al inicio del módulo, o modificando la opción equivalente en la ventana de propiedades del proyecto, conseguiremos que el lenguaje no distinga entre mayúsculas y minúsculas, de tal forma que `"Mañana"` sí sería igual a `"mañana"`.

Es posible utilizar en una misma expresión operadores aritméticos y relacionales, caso éste en el que, antes de que se evalúe una expresión relacional, previamente se realizarán los cálculos aritméticos necesarios para que los operandos a comparar sean sólo dos.

Operadores lógicos

En ocasiones una expresión relacional no está compuesta tan sólo de dos operandos y un operador de comparación, sino que son necesarios varios operadores y más operandos. En estos casos hay que unir las distintas subexpresiones

por medio de algún elemento, que fije a su vez unas reglas entre cada una de las relaciones. Entre otras esta es la finalidad de los operadores lógicos.

Los operadores lógicos, a excepción del operador `Not`, toman dos expresiones normalmente relacionales, que como sabemos pueden devolver `True` o `False`. Con esta pareja de valores el operador realiza otra operación, en este caso lógica, para obtener un sólo valor. En la tabla 5.3 puede encontrar los distintos operadores lógicos, así como los valores que devuelven según que las expresiones de entrada sean ciertas o falsas. **E1** representa el valor devuelto por la primera expresión y **E2** el devuelto por la segunda. Como ya se ha dicho estos valores sólo serán `True` o `False`.

Tabla 5.3. Operadores lógicos

Operador	Devuelve `True` si	Devuelve `False` si
`And`	E1=`True` y E2=`True`	En cualquier otro caso
`Or`	E1=`True` o E2=`True`	E1=`False` y E2=`False`
`Xor`	E1=`True` y E2=`False` o E1=`False` y E2=`True`	E1=`True` y E2=`True` o E1=`False` y E2=`False`
`Not`	E1=`False`	E1=`True`

Como podrá ver, el operador `Not` sólo toma un parámetro y lo que hace es devolver el valor inverso al devuelto por la expresión. Por ello se le llama operador de negación.

Aunque podemos utilizar estos operadores para obtener un resultado y almacenarlo en una variable de tipo `Bolean`, lo habitual, como ocurre con los operadores relacionales, es que los usemos en el contexto de una sentencia condicional o de control de repetición. Conocerá ese tipo de construcciones en el próximo capítulo.

Operadores de circuito corto

Una particularidad de los operadores lógicos de Visual Basic, que le diferencia de los equivalentes en la mayoría de los demás lenguajes, es que evalúan todas las subexpresiones siempre, incluso cuando no es necesario. Suponga que tiene la siguiente expresión condicional:

```
If 1 > 5 And Comprueba() Then
```

Está claro que 1 no es mayor que 5 por lo que, en la práctica, no sería necesario comprobar la subexpresión que hay detrás del operador `And`. Al ser `False` la primera el resultado será `False`, independientemente del valor que pudiera devolver la función `Comprueba()`. Visual Basic, sin embargo, llamaría a esa función. Puede comprobarlo con el código siguiente:

```
Sub Main()
```

```
  If 1 > 5 And Comprueba() Then

    Console.WriteLine( _
        "Es cierto, 1 > 5 y Comprueba() devuelve True")
  Else

    Console.WriteLine("1 no es mayor que 5")
  End If
End Sub


Function Comprueba() As Boolean
  Console.WriteLine("Comprueba() se ejecuta")
  Return True ' devolvemos True como resultado
End Function
```

Al ejecutar este programa obtendría un resultado como el de la figura 5.2, en el que podemos ver que la función `Comprueba()` se ha ejecutado, algo que en la mayoría de los lenguajes no ocurriría. Algo similar pasa con el operador `Or`. Analicemos la siguiente sentencia:

```
If 1 < 5 Or Comprueba() Then
```

En este caso quiere ejecutarse la sentencia que haya tras el condicional en caso de que 1 sea menor que 5 o bien `Comprueba()` devuelva `True`. Dado que la primera subexpresión devuelve `True`, 1 es menor que 5, no habría necesidad de evaluar la que hay detrás de `Or`, si bien Visual Basic la evalúa.

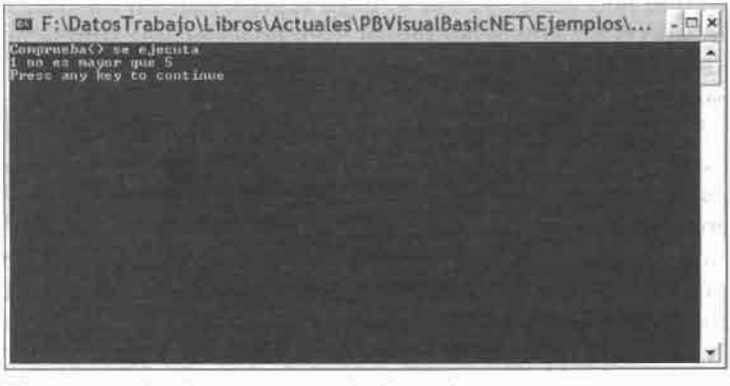


Figura 5.2. Podemos ver que la llamada a `Comprueba()` se produce

Visual Basic .NET cuenta también con dos nuevos operadores lógicos, llamados `AndAlso` y `OrElse`, que evalúan la expresión lógica utilizando lo que se conoce como *circuito corto*, que es el utilizado en la mayoría de los lenguajes. Usando `AndAlso`, si la subexpresión que hay a la izquierda es falsa la de la derecha no llega a evaluarse. Puede comprobarlo modificando el código anterior. Lo mismo ocurre con `OrElse`, que no evalúa la subexpresión que hay a la derecha si la de la izquierda es cierta.

Operaciones entre bits

Además de realizar operaciones lógicas, los operadores vistos en la tabla 5.3 también nos permiten manipular los bits individuales dentro de un número. En ocasiones, sobre todo cuando se realizan operaciones con funciones de bajo nivel, necesitamos comprobar si un cierto bit de un dato está encendido o apagado, o bien precisamos poner a 0 ó 1 un cierto bit. Para poder trabajar a nivel de bits es básico conocer la base 2 o binaria, cómo se pasa un número de decimal a binario o viceversa.

Partiendo de las reglas de cada uno de los operadores y asumiendo que un bit a 0 indica `False` mientras que un bit a 1 significa `True`, es fácil comprender el funcionamiento de las operaciones entre bits. Vamos a detallar seguidamente las tres actuaciones más habituales.

Si deseamos poner a 1 un bit de un determinado dato, llamémosle `N1`, utilizaremos el operador `Or`. Este tiene la característica de actuar sobre dos operadores y si cualquiera de ellos es `True` (1) el resultado es `True`. Por lo tanto, si tenemos `N1` y realizamos una operación lógica `Or` con el número `0` ¿qué ocurriría?; nada, `N1` quedaría igual, ya que si en él un bit está a `0` al realizar un `Or` con `0` obtenemos `0`, mientras que si un bit estaba a `1` al realizar un `Or` con `0` se obtendrá de nuevo `1`.

Partiendo de esta base vamos a suponer que `N1` contiene el conjunto de bits `00001001`, que equivale al número `9`, y deseamos activar el segundo bit[1]. Tendremos que realizar una operación lógica `Or` con un número cuyo segundo bit esté a `1` y el resto a `0`, de tal forma que no se modifique ningún bit, recuerde la operación que hemos realizado antes, excepto el que hemos puesto a `1`. El número que necesitamos es el `4`, cuya representación binaria, `00000100`, tiene a `1` tan sólo el bit que necesitamos. Pruebe a ejecutar simplemente la instrucción `Console.WriteLine(9 Or 4)`, obtendrá como resultado `13`, que pasado a decimal es `00001101`.

La operación contraria, poner un bit a `0`, la llevaremos a cabo con el operador que es funcionalmente contrario: `And`. Si tenemos un número, llamémosle `N1`, y realizamos una operación lógica `And` entre él y otro número cuyos bits estén todos a `1`, el resultado obtenido será de nuevo `N1`. Por lo tanto, para poner a `0` un cierto bit tendremos que hallar un número que tenga a `1` todos sus bits excepto aquel que deseamos poner a `0`. Siguiendo con el ejemplo anterior, supongamos que `N1` ahora tiene el valor `00001101`, tras la operación efectuada, y deseamos apagar el segundo bit. Necesitaríamos entonces la secuencia de bits `11111011`, que es la del número `251`. Ejecute a continuación la sentencia `Console.WriteLine(13 And 251)` y verá cómo obtiene el `9` que teníamos originalmente.

Por último tenemos que poder comprobar si un cierto bit está encendido o apagado, a `0` ó `1`. La técnica más fácil y común es poner a `0` todos los bits excepto el que se desea comprobar, si el número resultante es `0` es porque dicho

---

[1] Los bits se numeran de derecha a izquierda, desde el bit 0 hasta el bit 7. Por lo tanto en este caso los bits que están a 1 son el bit 0 y 3.

bit estaba a 0, si se obtiene otro número es porque estaba a 1. Volvamos una vez más sobre el ejemplo anterior. En este caso `N1` vuelve a tener el valor 9 y queremos comprobar si el segundo bit está apagado o encendido, por lo que deberemos apagar todos los bits menos ese. La máscara de bits necesaria es `00000100`, que corresponde de nuevo al número `4`. Ejecute ahora la sentencia `Console.WriteLine((9 And 4) <> 0)` y observe el resultado, `False`. Ello indica que la expresión es falsa, que el segundo bit no está a 1 en el número 9. Realice la misma prueba con el número 13 y verá como el valor devuelto es `True`, ya que en él sí está a 1 dicho bit.

Con el fin de aclarar algo más el concepto de trabajo con bits vamos a crear un pequeño programa cuya finalidad será mostrar la representación binaria de los números `0` a `255`. El resultado se muestra en la figura 5.3. El código necesario está estructurado en tres bucles, tal y como puede verse a continuación. Los dos primeros distribuyen la información en 64 filas por 4 columnas, el valor mínimo de `I` y `K` es `0`, siendo `I=0` y `K=0` la operación `I*4+K` devuelve `0`, mientras que el valor máximo es `255`, siendo `I=63` y `K=7` la misma operación devuelve `255`.

Cualquier número comprendido en este rango se puede representar con ocho bits, por ello existe un tercer bucle, controlado por la variable `J`, que recorre esos ocho bits pero en sentido inverso, de 7 a `0`, ya que como recordará los bits se numeran de derecha a izquierda.

Para saber si el número generado por `I*4+K` tiene a 1 un cierto bit, deberemos realizar una operación lógica `And` con un número que tenga a `0` todos los bits menos el que se desea comprobar, que es el bit `J`. Para crear dicho número se realiza la operación `2^J`, siendo `2` la base de numeración binaria y `J` el número de bit a comprobar.

Por lo tanto, si el resultado de la operación es `True` es que el bit está a 1 y así se imprime, mientras que en caso contrario es `0`.

```
Dim I, J, K As Integer

For I = 0 To 63
  Console.Write("{0}->{1}", I*4, vbTab)
  For K = 0 To 3
    For J = 7 To 0 Step -1 ' 1 1 1
```

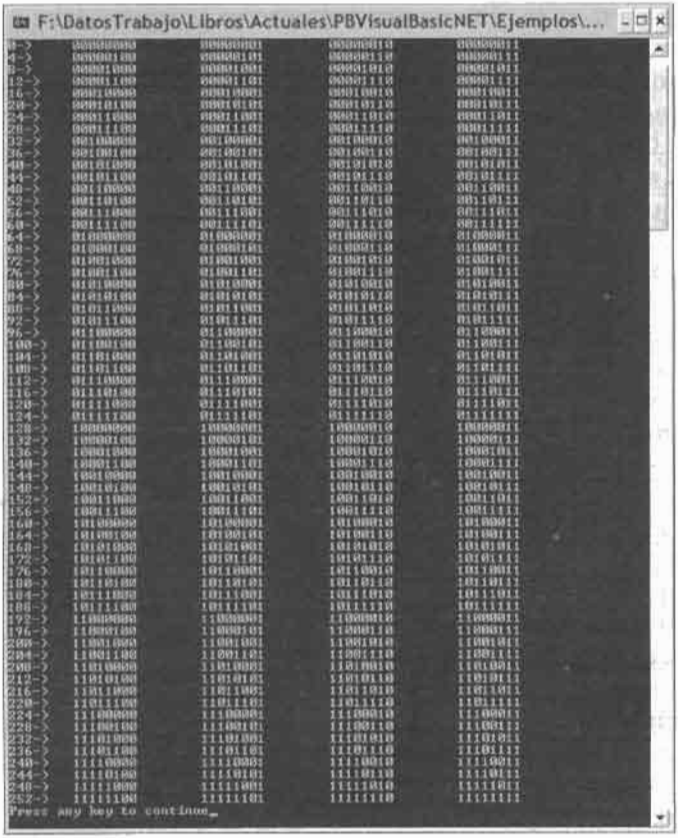


Figura 5.3. Representación binaria de los números 0 a 255

Expresiones con referencias

Los operadores relacionales, especialmente el operador = de igualdad, no pueden utilizarse siempre con todas las variables. En los ejemplos anteriores siempre hemos tratado con variables que almacenan valores, lo cual no plantea problema alguno. Al tratar con variables que mantienen referencias, sin embargo, el caso es distinto.

Aunque en Visual Basic es una posibilidad inexistente, otros lenguajes .NET, como es el caso de C # y C++, tienen la capacidad de facilitar una implementación a medida en sus clases para ciertos operadores, por ejemplo el de asignación e igualdad. Esto no implica, sin embargo, que todas las clases cuenten con una implementación de ese operador. Puede comprobarlo fácilmente con este código:

```
Dim Ventana1, Ventana2 As Object


Ventana1 = AppDomain.CurrentDomain
```

```
Ventana2 = Ventana1

Console.WriteLine(Ventana1.GetType.ToString())

Console.WriteLine(Ventana1 = Ventana2)
```

Al ejecutarlo verá que aparece el tipo de `Ventana1` pero, de inmediato, se produce una excepción (véase la figura 5.4). Lo que ocurre es que la clase `AppDomain` no dispone de una implementación del operador = para comprobar si dos referencias son iguales.

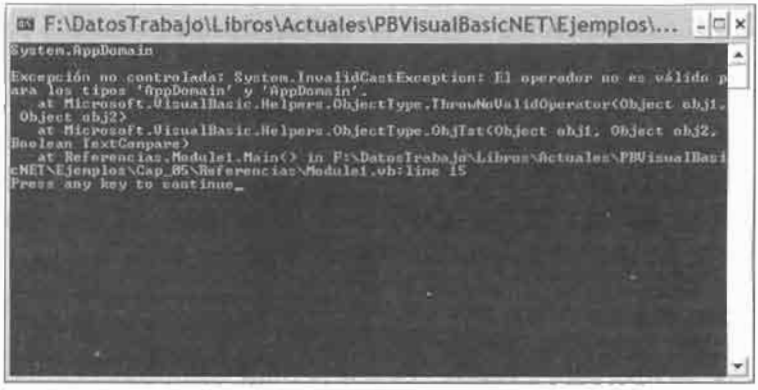


Figura 5.4. La comparación de las dos variables causa una excepción

La solución a esta necesidad, comprobar si dos variables tienen una referencia al mismo objeto, la encontramos en el operador `Is`. Basta con sustituir la expresión `Ventana1 = Ventana2` por `Ventana1 Is Ventana2` para obtener el resultado esperado.

Otro caso peculiar es el que se presenta, como veremos en el capítulo dedicado al trabajo con clases de objetos, cuando necesitamos comprobar el tipo de objeto al que mantenemos una referencia en una variable. Hay que tener en cuenta que hablamos del tipo del objeto, no del valor que contiene la variable, por lo que no son útiles operadores como = o `Is`. En este caso el operador a usar, lo utilizamos en un capítulo previo, sería `TypeOf Is`. En la sentencia siguiente, por ejemplo, puede ver cómo comprobamos si el tipo del objeto al que apunta `Ventana1` es `AppDomain`.

```
Console.WriteLine(TypeOf (Ventana1) Is AppDomain)
```

Puntos clave

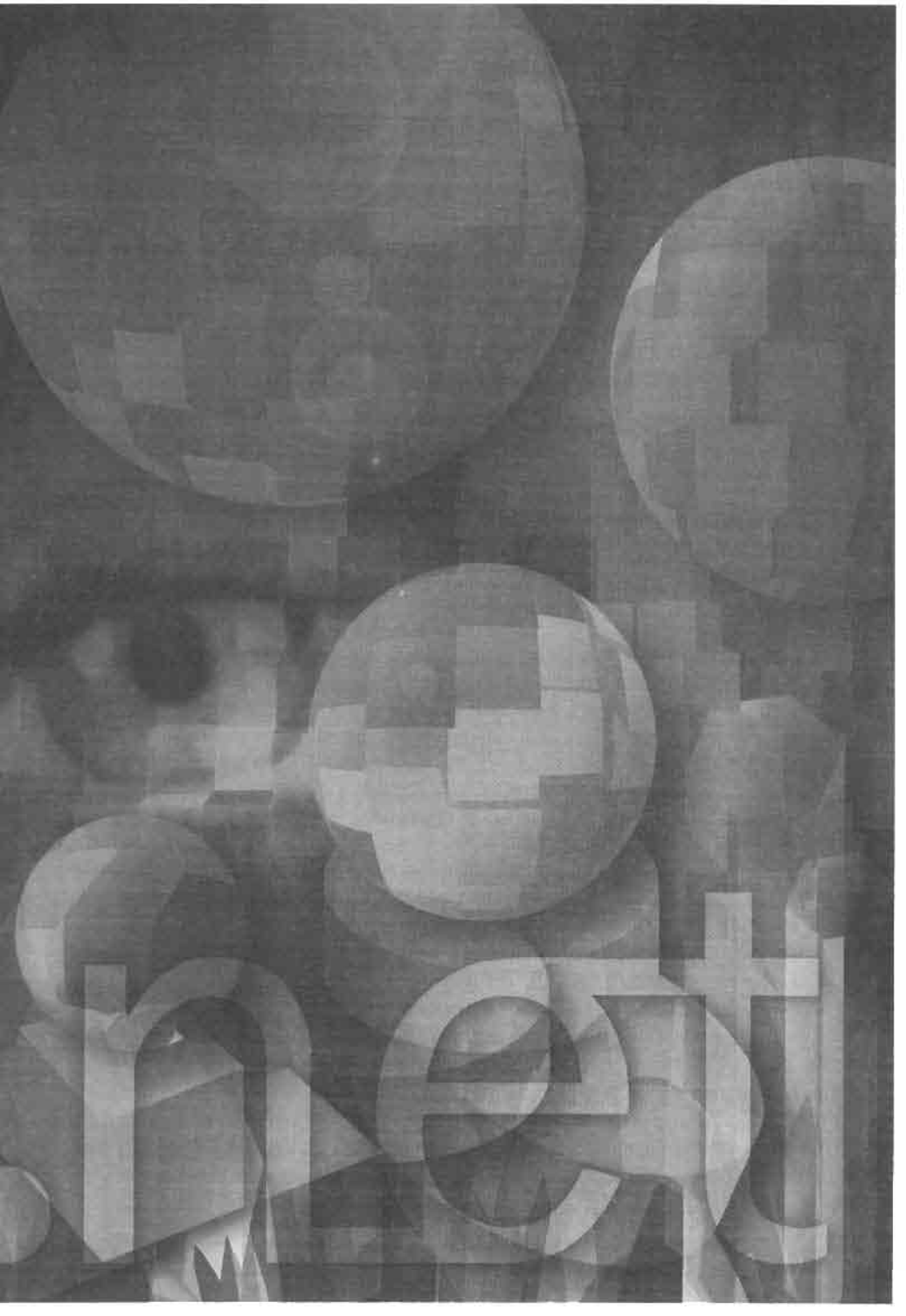
- Una expresión es una combinación de operandos y operadores que tiene por finalidad efectuar una o más operaciones obteniendo un resultado.

- Las expresiones simples pueden ser aritméticas, relacionales y lógicas, siendo habitual la combinación de estas categorías formando expresiones más complejas.
- Existen versiones abreviadas de operadores aritméticos que toman como primer operando la variable destinataria del resultado.
- Los operadores aritméticos facilitan las operaciones básicas: suma, resta, multiplicación, división, división entera, resto y potenciación.
- Mediante los operadores relacionales se establece una comparación entre los operandos, obteniendo como resultado `True` o `False` según que la relación sea cierta o no, respectivamente.
- Los operadores lógicos se utilizan, principalmente, para unir múltiples expresiones relacionales y obtener un resultado único.
- Visual Basic .NET dispone de dos nuevos operadores lógicos: `AndAlso` y `OrElse`. Son equivalentes a `And` y `Or` pero usan el sistema de evaluación de circuito corto.
- También podemos usar los operadores lógicos para manipular bits de un dato o comprobar el estado de esos bits.
- Para comprobar si dos variables mantienen una referencia a un mismo objeto debe utilizarse el operador `Is`, en lugar de `=`.
- El operador `TypeOf Is` nos permite comprobar el tipo del objeto correspondiente a la referencia almacenada en una variable.

Resumen

Ahora que conocemos los operadores disponibles en Visual Basic .NET podemos, en la práctica, efectuar cualquier operación sobre los datos de la aplicación: operaciones aritméticas, comprobaciones relacionales y composiciones lógicas. Muchas de estas expresiones tienen utilidad por sí mismas, por ejemplo una operación aritmética para obtener un resultado y almacenarlo en una variable, mientras que otras se aplican en diversas construcciones que tendrá ocasión de conocer en el próximo capítulo, como los condicionales y los bucles.

En la plataforma .NET existen ámbitos que contienen servicios mediante los cuales se facilitan operaciones más complejas. En el ámbito `System`, por poner un ejemplo, existe una clase, llamada `Math`, en la que encontrará multitud de funciones para realizar operaciones aritméticas y trigonométricas, como `Sin()`, `Cos()`, `Round()`, `Pow()` o `Sign()`. Otras clases, como `DateTime`, cuentan con métodos específicos para operar sobre el tipo de dato concreto que almacenan.



6
Estructuras de control

Tradicionalmente el lenguaje BASIC ha sido uno de los más pobres en cuanto a estructuras de control se refiere, disponiendo tan sólo de una sentencia para toma de decisiones, `If/Then/Else`, que no podía exceder de una línea y una sentencia para la ejecución de bucles, `For/Next`. Este hecho convirtió al lenguaje BASIC en uno de los menos estructurados y, por supuesto, muy alejado de la elegante estructuración del código de Pascal o C. La causa: instrucciones como `GoTo` o `GoSub`, que hacían que la ejecución saltase de un punto a otro del programa, dificultando su seguimiento y, en ocasiones, haciendo poco descifrable su funcionamiento.

Afortunadamente Visual Basic .NET dispone de muchas más estructuras de control que sus antepasados, como tendremos ocasión de ver en este capítulo. Estas nuevas estructuras afectan a sentencias ya existentes e incorporan otras nuevas, como el control estructurado de excepciones. Si usted ha programado anteriormente con otras versiones de BASIC que careciesen de estructuras de control, acostumbrado por lo tanto a los saltos de un punto a otro del programa, verá como en Visual Basic es posible escribir aplicaciones enteras sin utilizar un solo `GoTo`.

Estructuras condicionales

En muchas ocasiones las líneas de código de un método no se ejecutan de forma secuencial, sino que dependiendo de que la evaluación de una expresión

sea cierta o falsa se toman distintos caminos, o se ejecutan distintas instrucciones. Las sentencias que nos permiten realizar esta operación son las condicionales, basadas en la evaluación de expresiones en las que intervienen, por regla general, operadores relacionales y lógicos.

Para evaluar una expresión simple podemos utilizar la estructura `If/Then`, que ya hemos visto en algunos de los programas de los capítulos anteriores. Tras la sentencia `If` dispondremos una expresión a evaluar, que puede estar formada por operandos de cualquier tipo y operadores, tanto aritméticos como relacionales o lógicos. Como ya sabemos, cualquier expresión de este tipo se reduce a dos resultados: verdadero o distinto de `0`, en caso de que la expresión se cumpla, y falso o `0`, en caso de que no sea así. En el primer caso la sentencia que se haya dispuesto tras `Then` será ejecutada, en el segundo caso no se ejecutará. Por ejemplo, suponga que desea dar el valor `1` a una variable en caso de que otra tenga o no el valor `True`. La sentencia necesaria podría ser:

```
If Variable = True Then Resultado = 1
```

En este caso la expresión es `Variable = True`, una expresión relacional. En caso de que sea cierta, si la variable contiene ese valor, entonces se ejecutará la sentencia `Resultado = 1`. Esta expresión puede quedar aún más reducida, ya que `Variable`, asumiendo que es de tipo `Boolean`, puede contener sólo dos valores: `True` o `False`, y su valor puede ser utilizado directamente como expresión.

```
If Variable Then Resultado = 1
```

Obtendremos exactamente el mismo resultado, la expresión será cierta si `Variable` contiene el valor `True` y falsa si tiene `False`. Esta notación es utilizada muy a menudo para comprobar directamente si una variable tiene uno u otro valor.

En cualquier caso, tanto si la expresión se cumple como si no, la ejecución continuará por la siguiente línea de código. Es decir, la sentencia `If` utilizada de esta forma afecta sólo a la línea en la que se ha escrito la instrucción completa, pero no a las que le siguen.

Varias sentencias en una línea

En caso de que necesitemos ejecutar varias sentencias si la expresión se cumple, en el ejemplo anterior sólo se ejecuta una, podemos disponerlas una detrás de otra separándolas por dos puntos. Supongamos que además de darle el valor `1` a `Resultado` también quiera dar el valor `False` a la misma variable implicada en la decisión. La línea podría quedar como sigue:

```
If Variable Then Resultado = 1: Variable = False
```

Esta notación, el uso de los dos puntos para escribir varias sentencias en una línea, no es exclusiva de las estructuras de control, puede utilizarla en

cualquier momento, en cualquier línea de código. Sin embargo esta práctica hace que el código sea menos legible, procediendo de los días en que los programas BASIC debían estar distribuidos en líneas numeradas, en cada una de las cuales se disponían varias sentencias.

La salida falsa

Como se ha dicho, tanto si la expresión es cierta, y por lo tanto se ejecutan las sentencias que hay tras `Then`, como si no lo es, la ejecución del programa seguirá por la línea de código que corresponda. Por lo tanto habremos conseguido que esas sentencias se ejecuten sólo si la expresión es cierta y en ningún otro caso.

En caso de que deseemos que otra sentencia o sentencias se ejecuten sólo si la expresión no es cierta podemos utilizar la palabra `Else`, tras la cual dispondremos esa sentencia o sentencias. Por ejemplo, observe este fragmento de código:

```
Console.WriteLine("Se ejecuta siempre (antes)")
If Variable Then Console.WriteLine("Es cierto") Else _
   Console.WriteLine("Es falso")
Console.WriteLine("Se ejecuta siempre (después)")
```

La primera línea se ejecutará siempre, la segunda contiene una decisión dependiendo de la cual se ejecuta una sentencia u otra, mientras que la tercera línea se ejecutará en cualquier caso. Fíjese en que si hubiésemos dispuesto la llamada `Console.WriteLine("Es falso")` en la línea siguiente al `If`, se ejecutaría siempre, mientras que tal y como se han estructurado sólo lo hará si la expresión es falsa, en ningún caso más.

Nota

Tenga en cuenta la diferencia entre líneas físicas de código y líneas lógicas. En el fragmento de código anterior sólo hay tres líneas de programa, aunque físicamente una de ellas ha tenido que partirse en dos. Observe la existencia del carácter _ justo detrás de la palabra `Else`. Ella indica que la línea siguiente es, en realidad, continuación de ésta.

Condiciones anidadas

Se puede dar el caso de que la sentencia que sigue a un `Then` sea otro `If`, o que la sentencia que sigue a un `Else` sea otro `If`, y esta circunstancia puede repetirse varias veces de tal forma que tengamos una serie de decisiones anidadas. En este caso la notación de las sentencias `If ... Then ... If ... Then ... Else ... If ...` puede ser poco menos que críptico y puede llevar a error, ejecutándose unas instrucciones cuando se deberían ejecutar otras. En este caso es mejor utilizar el segundo formato de la sentencia `If`, mediante

la cual podemos distribuir el código sujeto a una estructura de decisión en varias líneas. El último ejemplo expuesto anteriormente quedaría así de claro.

```
Console.WriteLine("Se ejecuta siempre (antes)")
If Variable Then
  Console.WriteLine("Es cierto")
Else
  Console.WriteLine("Es falso")
End If
Console.WriteLine("Se ejecuta siempre (después)")
```

En este caso queda bastante más claro qué líneas se ejecutarán en un caso, en otro o en cualquier caso. Todo el código comprendido entre `If` y `End If` forma parte de la estructura de decisión, mientras que el que queda fuera se ejecutará independientemente del resultado de la expresión. Utilizando esta nueva notación el código existente en varias decisiones anidadas queda dispuesto de una forma mucho más clara, como puede ver seguidamente.

```
Dim A, B, C As Integer
If A = 1 Then
  If B = 5 Then
    C = A + B
  Else
    C = A - B
  End If
ElseIf A = 3 Then
  C = A * B
Else
  C = A / B
End If
```

Fíjese cómo la parte que se ejecutará si `A=1` contiene otra decisión, con su correspondiente `If ... End If`. La parte falsa, si `A` no es `1`, es también una decisión, por lo que se usa la unión `ElseIf` para componerla. A la derecha de cada asignación se detallan los valores que toman `A` y `B` para que se ejecute esa línea y no las otras.

Nota

Si intenta ejecutar este código observará que se produce una excepción. Esto se debe a que las variables `A` y `B`, dado que no se les ha asignado un valor, contienen `0`, lo cual causa que se evalúe la expresión `C = A / B`, generando una división por cero. Más adelante, en este mismo capítulo, aprenderá a controlar estas excepciones.

Condiciones que siempre devuelven un valor

Si en nuestro código la sentencia condicional tiene como única finalidad asignar un valor u otro dependiendo del resultado de la expresión, podemos

utilizar la función `IIf()`. Esta toma tres parámetros: la expresión a evaluar, el valor a devolver si se cumple y el valor a devolver si no se cumple. Por ejemplo, imagine que desea dar a la variable A el valor 1 ó 2, dependiendo de que B sea 5, por poner un ejemplo.

```
If B = 5 Then A = 1 Else A = 2

A = IIf( B = 5, 1, 2 )
```

En la primera línea utilizamos la estructura `If/Then`, mientras que en la segunda usamos la función `IIf()`. En ambos casos el resultado es el mismo.

El ejemplo anterior es el caso más simple, ya que tan sólo existe una expresión a evaluar y dos valores a devolver. Pero imagine que el valor que debe recibir A depende de más expresiones, tal y como puede ser la siguiente estructura de decisión:

```
If B = 5 Then
      A = 1
ElseIf B <= 0 Then
      A = -1
ElseIf B > 5 Then
      A = 0
End If
```

Toda esta secuencia de decisiones puede reducirse a tan sólo una línea por medio de la función `Switch()`. Esta función puede tomar dos o más parámetros, siempre por parejas. El primer elemento de cada pareja es la expresión a evaluar, mientras que el segundo corresponde al valor a devolver en caso de que la expresión sea cierta. La primera expresión que se cumpla devolverá el valor correspondiente. El código anterior quedaría reducido a la siguiente línea, obteniéndose exactamente el mismo resultado.

```
A = Switch(B = 5, 1, B <= 0, -1, B > 5, 0)
```

En caso de que las expresiones sean aún más elementales y se desee simplemente devolver un valor u otro dependiendo de una variable o expresión utilizada como índice, como puede ser el caso siguiente:

```
If B = 1 Then
      A = 5
ElseIf B = 2 Then
      A = 7
ElseIf B = 3 Then
      A = 11
EndIf
```

podemos obviamente simplificar este código utilizando la función `Switch()`, que quedaría como:

```
A = Switch( B = 1, 5, B = 2, 7, B = 3, 11 )
```

Pero aún puede quedar más reducido utilizando la función `Choose()`. Esta toma un primer parámetro que actuará como índice y después tantos valores como sea necesario, de tal forma que dependiendo del valor del índice se devuelva un valor u otro. Nuestra expresión original de siete líneas quedaría tan simple como puede ver a continuación.

```
A = Choose( B, 5, 7, 11 )
```

Al utilizar tanto `Switch()` como `Choose()` tenga en cuenta que si no es posible devolver ningún valor, porque no se cumpla ninguna de las expresiones, en el primer caso, o porque el índice sea cero o mayor al número de valores existentes, en el segundo, estas funciones devolverán el valor cero.

Condicionales múltiples

Hemos visto como el uso de las funciones `Switch()` y `Choose()` pueden simplificar el código de forma llamativa, pero sólo las podemos utilizar cuando dependiendo de la expresión simplemente devolvamos un valor. Imagine el último ejemplo expuesto, que `B` pueda tomar los valores `1`, `2` o `3`, y que dependiendo de ello tenga, no que devolver un valor, sino que ejecutar unas u otras sentencias. En este caso ni `Switch()` ni `Choose()` nos serán útiles y tendríamos que utilizar la estructura que vimos al principio de `If/Then/ElseIf/End If`.

En casos como éste nos será muy útil la estructura condicional `Select Case`, que nos permite evaluar una expresión y a continuación enumerar posibles resultados, de tal forma que se ejecute un grupo u otro de sentencias dependiendo del resultado que coincida con el obtenido por la expresión. Como ejemplo supongamos que dependiendo del valor de `B` tengamos que imprimir un texto u otro y, además, asignar un determinado valor a `A`.

```
Select Case B
  Case 0
    Console.WriteLine("El valor de B no es valido")
    A = 0
  Case 5
    Console.WriteLine("B tiene el valor óptimo")
    A = 10
  Case Is < 5
    Console.WriteLine("B está por debajo del valor optimo")
    A = 3
  Case Is > 5
    Console.WriteLine("B está por encima del valor optimo")
    A = 7
End Select
```

En el bloque de código anterior primero se compara si `B` es `0`, de ser así se imprime un mensaje determinado y se le da a la variable `A` el valor `0`. A continuación se compara con otra constante, en este caso `5`. El tercer `Case` corresponde a la expresión `B < 5`, es decir, en el caso de que `B` sea menor que `5` se

ejecutarán las dos líneas que hay en ese bloque. Por último se ve si B es mayor que 5.

En este ejemplo tenemos un caso por cada bloque. El mensaje "El valor de B no es válido" sólo se imprimirá si B es 0 y no en otro caso. Imagine, sin embargo, que esa misma actuación se deba realizar si B es 10. Una solución sería utilizar dos Case, uno con el valor 0 y otro con el valor 10, escribiendo en ellos las mismas sentencias. Esto, obviamente, nos llevaría a la repetición de un mismo código en distintos sitios. Para evitarlo podemos utilizar dos notaciones distintas, ambas válidas, que puede ver en la siguiente porción de código.

```
Case 0
Case 10
      Console.WriteLine("El valor de B no es válido")
      A = 0
--------------------
Case 0, 10
  Console.WriteLine("El valor de B no es válido")
  A = 0
```

De igual forma podemos enumerar tres, cuatro o cualquier otro número de casos para los que exista una única solución. En caso de que varios de estos casos sean secuenciales, por ejemplo deseemos ejecutar un cierto código si B es 2, 3, 4 ó 5, en lugar de utilizar Case 2,3,4,5 podemos usar la notación Case 2 To 5.

También podemos utilizar operadores lógicos en uno de los apartados de un Select Case, hecho que nos permite aún una mayor flexibilidad ya que si tras la sentencia Select Case tan sólo podemos disponer una expresión evaluable a un segundo valor, después en cada Case es posible componer expresiones más complejas. Por ejemplo, imagine que desea ejecutar un cierto código si B contiene 3, 4, 5, 6 u 8 y, además, la variable C no es cero.

```
Select Case B
  Case 3 To 6, 8 And C
      Console.WriteLine("La condicion se cumple")
      ...
```

Puede darse el caso de que el resultado obtenido de la expresión no coincida con ninguno de los Case que nosotros hemos dispuesto, caso en el que no se ejecutará ningún código existente entre Select Case y End Case a no ser que exista el apartado Case Else. Cualquier código que pongamos a continuación de esta opción se ejecutará siempre que no se cumpla ninguna de las otras condiciones. Por ejemplo, en el siguiente fragmento de código el mensaje "Valor no válido para B" se mostrará cuando B no sea 5, no esté entre 1 y 4 y no sea mayor que 5, por lo que los casos posibles son que B sea 0 o bien negativo.

```
Select Case B
  Case 5
    Console.WriteLine("B tiene el valor óptimo")
```

```
        A = 10
      Case Is > 0 And B < 5
        Console.WriteLine("B está por debajo del valor óptimo")
        A = 3
      Case Is > 5
        Console.WriteLine("B está por encima del valor óptimo")
        A = 7
      Case Else
        Console.WriteLine("Valor no válido para B")
        A = 0
    End Select
```

Estructuras de repetición

Durante la ejecución de un programa, asumiendo que cuenta con una interfaz de usuario, el código que nosotros escribimos se está repitiendo constantemente. Cada vez que el usuario selecciona una opción, pulsa un botón, usa el teclado, etc., el mismo código es ejecutado para responder a los mismos eventos. Esta repetición se efectúa a demanda del usuario, cuando es necesario, pero en ocasiones es necesario repetir las mismas líneas de código un número determinado de veces seguidas. Obviamente no vamos a escribir múltiples veces el mismo código, sino que crearemos un bucle sirviéndonos de algunas de las sentencias con que cuenta Visual Basic para ello.

Bucles por contador

El tipo de bucle más conocido en BASIC es el formado por las instrucciones `For/Next`. La primera pone en marcha el bucle, inicializando una variable con un valor de origen, fijando un valor de fin y opcionalmente un incremento en cada vuelta del bucle. `Next` marca el final de la sección de código que se desea repetir. A este bucle se le llama bucle por contador porque utiliza una variable numérica a la que se va sumando o restando un valor en cada vuelta del bucle, cuando el contenido de la variable llega a un límite el bucle se termina. Suponga que desea imprimir en el formulario los números `10` a `20` ambos incluidos, bastará con el siguiente bucle.

```
Dim Número As Byte
For Número = 10 To 20
  Console.WriteLine(Número)
Next
```

En este caso nosotros no hemos especificado ningún tipo de incremento, por lo que se ha asumido que había que ir sumando uno a la variable en cada vuelta. Este incremento puede ser modificado utilizando `Step` seguido del valor, que puede ser positivo, negativo e incluso cero, caso éste en que al no sumársele ningún valor a la variable el bucle puede ser un bucle sin fin, a no ser que desde el interior el valor de la variable se modifique por algún proceso.

Por ejemplo, suponga que desea imprimir los números pares del 10 al 20, el código necesario sería el que se muestra a continuación.

```
Dim Número As Byte
For Número = 10 To 20 Step 2
  Console.WriteLine(Número)
Next
```

Como verá, la instrucción `Next` no lleva ningún parámetro, aunque opcionalmente es posible poner a continuación el nombre de la variable que se está utilizando como contador del bucle. Si se usa cualquier otro identificador obtendremos un error. Habitualmente no se pone el nombre de la variable tras `Next` a no ser que creemos varios bucles anidados unos dentro de otros, caso éste en que el nombre de la variable aclara qué bucle es el que se termina.

Un bucle que está programado para ejecutarse un número determinado de veces no tiene por qué completar siempre su ejecución, puede darse el caso de que en el interior del bucle exista un proceso por el que en un momento determinado interese darlo por terminado. Para ello disponemos de la sentencia `Exit For`, que provocará la salida inmediata del bucle. En el siguiente fragmento de código se evalúa una expresión en el interior de un bucle, cuando ésta sea verdadera se provocará la salida.

```
Dim Número As Byte
For Número = 10 To 20 Step 2
  If Número = 16 Then Exit For
  Console.WriteLine(Número)
Next
```

Bucles condicionales

Un bucle no siempre es tan simple como para estar controlado por un contador, a veces no es posible saber de antemano el número de veces que el código se debe de ejecutar. En estos casos utilizaremos los bucles condicionales, que nos permiten repetir la ejecución de una porción de código *mientras* una cierta condición sea cierta o *hasta* que una cierta condición sea cierta.

La estructura `Do While` **condición**/`Loop` nos permite realizar un bucle que se repetirá mientras la condición especificada sea cierta, o dicho de otra forma, el bucle se terminará cuando la condición sea falsa. Por ejemplo, suponga que necesita pedir un dato que no puede ser una cadena vacía, obligatoriamente debe tener al menos un carácter. El siguiente fragmento de código hace precisamente eso, presentar continuamente un mensaje de saludo y mientras el usuario no conteste con algo el proceso se repetirá una y otra vez.

```
Dim Saludo As String
Do While Saludo = ""
  Console.WriteLine("Hola, ¿qué tal?")
  Saludo = Console.ReadLine()
Loop
Console.WriteLine(Saludo)
```

En este caso la variable `Saludo` se acaba de declarar, por lo que su valor inicial es la cadena vacía. Eso significa que al entrar en el bucle la condición inicialmente es cierta. Pero suponga que, antes de entrar en el bucle, `Saludo` ya tiene algún valor. El bucle no llegará a ejecutarse ninguna vez, puede probarlo simplemente añadiendo la línea `Saludo = "Hola"` justo después de la declaración. En los casos en que interese que el bucle se ejecute al menos una vez, aunque la condición no sea cierta en principio, bastará con desplazar la condición a la parte final del código, pasando así de un bucle en el que la evaluación de la expresión se efectúa a la entrada, a otro en el que la expresión se evalúa a la salida. El bucle quedaría como se muestra a continuación.

```
Do
  Console.WriteLine("Hola, ¿qué tal?")
  Saludo = Console.ReadLine()
Loop While Saludo = ""
```

Si en los anteriores bucles cambiamos la palabra `While` por `Until`, indistintamente de que la evaluación se efectúe a la entrada o a la salida del bucle, conseguiremos negar la condición, de tal forma que el bucle ahora no se ejecutaría mientras la condición fuese cierta, si no que lo haría mientras fuese falsa. Por ejemplo, suponga que desea crear un bucle en el que se vayan solicitando datos hasta que el usuario introduzca la cadena `Fin`, momento en que se dará por terminado el proceso. El bucle que necesitamos es el que se muestra en el siguiente fragmento.

```
Do Until Saludo = "Fin"
  Console.WriteLine("Hola, ¿qué tal?")
  Saludo = Console.ReadLine()
Loop
```

Todas las expresiones que se han mostrado en los ejemplos son simples, comprobando sólo una relación entre dos operandos. Por supuesto podemos utilizar expresiones más complejas en las que se vean envueltos operadores lógicos, relacionales y aritméticos.

De igual forma que era posible terminar un bucle por contador desde el interior del propio bucle, con `Exit For`, para los bucles por condición disponemos de la sentencia `Exit Do`, que dará por finalizado el bucle pasando la ejecución a la siguiente línea de código posterior al `Loop`.

Bucles que recorren un arreglo

A menudo cuando utilicemos un bucle será para trabajar sobre los elementos de un arreglo, asignándole un valor, realizando cálculos, etc. El siguiente fragmento de código, por ejemplo, crea un arreglo de cadenas y pide valores para cada elemento.

```
Dim Cadenas(9) As String
Dim I As Integer
```

```
For I = 0 To 9
  Console.WriteLine("Texto para el elemento {0}: ", I)
  Cadenas(I) = Console.ReadLine()
Next
```

Aquí `Cadenas(I)` representa a cada uno de los elementos del arreglo, ya que `I` irá recorriendo los índices existentes, en este caso desde el 0 hasta el 9. A continuación podría imprimir cada uno de los valores que se han introducido, con un bucle del mismo tipo. También podemos utilizar el bucle `For Each .. In`, cuya finalidad es recorrer todos los elementos de un arreglo o una colección, sin necesidad de indicarle índices ni valores mínimos o máximos. Entre las palabras `Each` e `In` deberemos insertar una variable, habitualmente de tipo `Object`, que irá tomando el valor de cada uno de los elementos que existan en el arreglo, ejecutando el bucle por cada uno de ellos. Para imprimir el contenido del arreglo el código necesario sería el siguiente. En él la variable `Elemento` tomaría el valor de `Cadenas(0)` y se ejecutaría el bucle, a continuación el valor de `Cadenas(1)` y se repetiría el proceso hasta llegar al último elemento.

```
Dim Cadena As String
For Each Cadena In Cadenas
  Console.WriteLine(Cadena)
Next
```

Nota

En realidad, podemos usar la construcción `For Each/Next` con cualquier tipo de dato que implemente la interfaz `System.IEnumerable`, no sólo con colecciones y arreglos. En un capítulo posterior conocerá las interfaces, su implementación y uso.

Utilizar este tipo de bucle en lugar de un bucle por contador tiene algunas ventajas. Una de ellas es que si después del diseño original del programa éste se modifica y el número de elementos del arreglo se reduce o se amplía, en un bucle por contador habría que retocar los índices, mientras que un bucle `For Each` esto no sería necesario ya que se recorren los elementos que existan en ese momento.

Control estructurado de excepciones

Visual Basic era, hasta ahora, uno de los pocos lenguajes que carecían de los elementos necesarios para controlar excepciones de una forma estructurada, sin necesidad de estar comprobando códigos de retorno de error o saltando de un punto a otro del programa. Visual Basic .NET introduce como otra novedad el control estructurado de excepciones, una novedad que nos permitirá sustituir el acostumbrado `On Error Goto` por otra construcción más flexible.

La estructura general para controlar una posible, o posibles, excepción es la siguiente:

```
Try

Catch TipoExcepcion

Catch TipoExcepcion

Finally

End Try
```

Las sentencias que aparecen después de la palabra `Try` son las que, potencialmente, podrían causar una excepción durante la ejecución del programa. Se ejecutarán normalmente a menos que aparezca dicha excepción, caso en el cual se transferirá el control a las sentencias del `Catch` apropiado, dependiendo del tipo de excepción. Por último, las sentencias que hay tras la palabra `Finally` se ejecutarán siempre, indistintamente de que se produzca o no una excepción. Este apartado, que es opcional, es usado generalmente para liberar algún tipo de recurso.

Observe el código siguiente, en él se genera una excepción al dividir un número por cero. La excepción es capturada y se traduce en la salida de un mensaje. Si no hubiésemos utilizado `Try/Catch` la excepción detendría la ejecución del programa comunicando el error.

```
Dim Cociente, Divisor As Integer

Try
    Cociente = 25 \ Divisor
Catch
    Console.WriteLine( _
      "Se ha producido una división por cero")
End Try
```

La sentencia que hay tras la palabra `Catch` se ejecutará siempre que se genere una excepción en algunas de las líneas que siguen a `Try`, en este caso sólo una. Si nos pusiésemos en un caso algo más complejo, en el que las sentencias a controlar pudieran generar distintas excepciones, podríamos usar una variante de `Catch` para efectuar un filtrado. Tomemos como base el fragmento de código mostrado a continuación.

```
Dim Cociente As Byte
Dim Dividendo, Divisor As Integer

Try
   Console.WriteLine("Introduzca el dividendo: ")

   Dividendo = CInt(Console.ReadLine())
   Console.WriteLine("Introduzca el divisor: ")
```

```
Divisor = CInt(Console.ReadLine())

Cociente = Dividendo \ Divisor

Console.WriteLine("El resultado es {0}", Cociente)
```

Las excepciones que pueden producirse en este caso son varias, aparte de la división por cero. Observe que la variable `Cociente` es de tipo `Byte` y no `Integer`, de tal manera que si el resultado de la división es superior a 255 se produciría un desbordamiento. Al convertir el valor solicitado por teclado a número, mediante la función `CInt()`, también se podría generar una excepción si dicha conversión fallase, por ejemplo por haber introducido un dato no numérico. Por último, cabe la posibilidad de que se diera otro tipo de error aritmético.

Cada una de las excepciones posibles está representada en la plataforma .NET por una clase.

Puede utilizar el Examinador de objetos, como se muestra en la figura 6.1, para localizar las distintas clases de excepciones. Todas estas clases están derivadas de una base común: la clase `Exception`, que puede ver seleccionada en el panel izquierdo de la figura 6.1. En el panel derecho de esa misma figura se enumeran los miembros con que cuenta esta clase, algunos de los cuales alojan información que puede resultarnos útil.

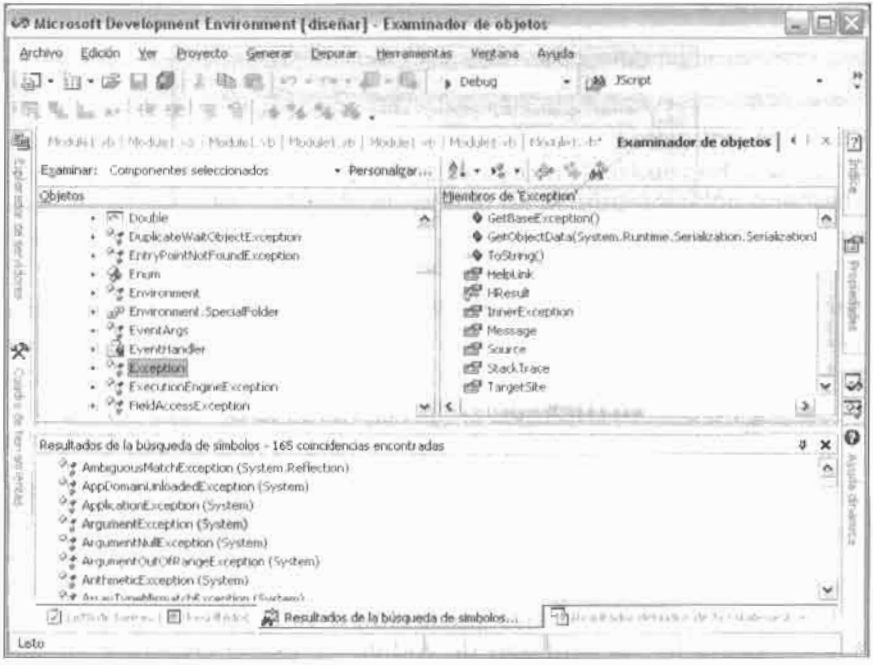


Figura 6.1. Buscamos las clases de excepciones utilizando el Examinador de objetos

Conociendo algunas de las clases de excepciones existentes, al fragmento de código anterior podríamos añadirle el siguiente para controlar tres de las categorías de excepciones que, según comentábamos antes, podrían producirse.

```
Catch X As DivideByZeroException
   Console.WriteLine( _
       "Se ha producido una división por cero")

Catch X As OverflowException
   Console.WriteLine( _
       "Se ha producido un desbordamiento")

Catch X As ArithmeticException
   Console.WriteLine( _
       "Se ha producido otro error aritmético")
   Console.WriteLine(X.Message)
   Console.WriteLine(X.ToString)
End Try
```

La `X` que aparece tras cada `Catch` es una variable de la clase de excepción puesta a continuación. En lugar de llamarla `X` podría tener cualquier otro nombre, ya que es un identificador creado por nosotros. El control del programa se transferirá a uno u otro `Catch` dependiendo de la excepción que se produzca.

Si introduce un valor no numérico como dividendo o divisor dará lugar a una excepción no controlada, por lo que aparecerá una ventana como la de la figura 6.2 comunicando el error.

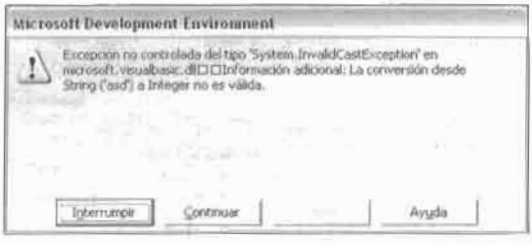


Figura 6.2. Aviso de una excepción incontrolada

Procedimientos y funciones

Hasta ahora todo el código que hemos necesitado para que nuestros programas funcionen lo hemos insertado, por regla general, en el método `Main()` que existe por defecto en el módulo, porque en realidad se ha tratado de llevar a cabo operaciones bastante sencillas. Sin embargo, cuando el programa tenga alguna finalidad más que la simple comprobación o prueba de un control o un operador, la cantidad de código necesaria crecerá y en ocasiones necesitaremos el mismo código en distintos puntos del programa. Obviamente la solución no es introducir una y otra vez todo el código, tanto si se repite como si

no. Para evitarlo podemos crear nuestros propios procedimientos y funciones, introduciendo en ellos el código que después podrá ser utilizado con una simple llamada.

La única diferencia existente entre un procedimiento y una función es que el primero no puede devolver parámetro alguno, mientras que la segunda sí. Por lo tanto, si vamos a codificar un proceso que realiza una determinada acción pero no devuelve valor alguno lo definiremos como un procedimiento, pero si es necesario devolver un valor cualquiera lo definiremos como función. Salvo que se indique lo contrario, todo lo que a continuación se aplique a un procedimiento se aplicará asimismo a una función. La denominación *método* se aplica a procedimientos y funciones que forman parte de una clase. `WriteLine()`, por ejemplo, es un método de la clase `Console`.

Declaración de un procedimiento

Básicamente, podemos crear un procedimiento en el interior de un módulo, como miembro de una estructura o como parte de una clase. Esa clase puede ser, por ejemplo, un formulario Windows, un componente o un servicio Web. Los procedimientos, al igual que las variables, pueden ser públicos o privados, según puedan ser utilizados sólo desde el módulo en que se han definido o desde cualquier otro módulo de la aplicación. Por defecto los procedimientos son públicos, por lo que no es necesario usar ningún modificador si es ésa la visibilidad que deseamos. En caso contrario podemos usar `Private`, `Protected` o `Friend`.

Nota

Profundizaremos sobre los ámbitos o visibilidad de los métodos, analizando el uso de modificadores tales como `Private`, `Protected`, `Public` y `Friend`, en el capítulo dedicado al estudio de la programación orientada a objetos.

Sabemos que todas las variables que declaremos en el interior de un procedimiento serán variables locales, cuya vida comienza al entrar en el procedimiento y termina al salir de él, perdiéndose por lo tanto el valor que contenga. Para evitar esta pérdida podíamos crear variables estáticas, tal y como vimos en un capítulo previo.

Los nombres de los procedimientos, que deberemos poner tras la palabra `Sub`, siguen las mismas reglas que los de variables, estructuras, clases e interfaces, pudiendo estar formados por letras, números y caracteres de subrayado.

Recepción de parámetros

Muchos de los procedimientos que diseñemos en nuestros programas estarán destinados a realizar algún tipo de operación, generalmente partiendo de

datos que se facilitan en el momento de la llamada. Estos parámetros deben estar enumerados entre los paréntesis que siguen al nombre del procedimiento, indicando su nombre y su tipo. Por ejemplo, suponga que va a crear un procedimiento cuya finalidad es colocar un texto en unas coordenadas específicas, por lo que necesita recibir tres parámetros: la cadena de texto y dos enteros con las coordenadas. La definición podría quedar como:

```
Public Sub Imprime(ByVal Cadena As String, ByVal X As Integer, _
                   ByVal Y As Integer)

End Sub
```

De igual forma que hemos trabajado con cadenas y enteros, es posible pasar parámetros prácticamente de cualquier tipo, desde estos tipos básicos hasta objetos, arreglos e incluso tipos definidos por nosotros.

Si se pasa como parámetro un arreglo, no importa de que tipo, en la declaración pondremos tras el nombre de la variable unos paréntesis vacíos, lo que nos permitirá recibir cualquier número de elementos sin conocerlo de antemano. El código que envía el arreglo como parámetro usará sólo el nombre de la variable, sin paréntesis y sin índices. Por ejemplo, en el siguiente fragmento de código puede ver un procedimiento que recibe un arreglo de cadenas, obtiene sus límites, inferior y superior, e imprime cada uno de los elementos.

```
Private Sub MuestraElementos(ByVal Nombres() As String)
  Dim I As Integer
  For I = Nombres.GetLowerBound(0) To Nombres.GetUpperBound(0)
    Console.WriteLine(Nombres(I))
  Next
End Sub
```

En cualquier punto en el que tengamos un arreglo de cadenas, y necesitemos imprimirlo, no tenemos mas que llamar a `MuestraElementos()` pasando como parámetro el nombre del arreglo. Por ejemplo, estas líneas de código inicializan una matriz con una serie de cadenas y después llaman al procedimiento anterior para imprimirlas.

```
Dim X() As String = { _
  "Uno", "Dos", "Tres", "Cuatro", "Cinco", "Seis"}

MuestraElementos(X)
```

Devolución de valores

En caso de que nuestro procedimiento deba devolver cualquier valor al código que le llamó, entonces tendremos que definirlo como una función. Esta puede ser tanto privada como pública y toma los parámetros de igual modo que un procedimiento. Tras el cierre de paréntesis, sin embargo, tendremos que añadir la palabra `As` y el tipo de dato que devolverá la función, aparte de cambiar la palabra `Sub` por `Function`. Por ejemplo, supongamos que vamos a

escribir una función que devuelva la suma de los cuadrados de dos números, devolviendo un `Long` con el resultado. La cabecera de la función sería la siguiente:

```
Private Function SumaCuadrados( _
   ByVal N1 As Integer, ByVal N2 As Integer) As Long
```

Una vez que nuestra función haya realizado las operaciones pertinentes y disponga del resultado, para devolverlo solamente ha de utilizar la sentencia `Return` disponiendo el valor detrás. Nuestra función `SumaCuadrados()` podría quedar como se muestra a continuación.

```
Private Function SumaCuadrados(ByVal N1 As Integer, _
                               ByVal N2 As Integer) As Long
  If N1 = 0 OrElse N2 = 0 Then
    Return -1
  Else
    Return N1 * N1 + N2 * N2
  End If
End Function
```

Aquí puede ver cómo la función devuelve el valor –1 si cualquiera de los parámetros es 0 o el resultado de la operación. Éste nunca podría ser –1, ya que el cuadrado de un número, aunque sea negativo, siempre es positivo y la suma de dos números positivos no puede ser negativa. Puede comprobar su funcionamiento con una simple sentencia como `Console.WriteLine(SumaCuadrados(2,5))`.

Devolución de arreglos

En caso de que una función deba devolver un arreglo, la notación a usar es similar a la utilizada para indicar que un parámetro a recibir es un arreglo. Es decir, dispondremos detrás del tipo unos paréntesis vacíos, sin especificar el número de elementos.

La posibilidad de devolver un arreglo desde una función es bastante interesante, ya que se pueden facilitar tareas que de otra forma serían bastante más complejas. Existe una función que devuelve un arreglo: `Split()`. Ésta toma una cadena de caracteres y genera un arreglo de tipo `String`, asignando a cada elemento una porción de la cadena original delimitada por un cierto separador. La función `Split()`, sin embargo, no es útil si precisamos, por ejemplo, crear un arreglo `Integer` a partir de una serie de datos independientes.

Sabiendo que cualquier función propia puede devolver arreglos, codificar un método que permita generar uno a partir de una serie de números facilitados como parámetros no parece demasiado complejo. Los datos de entrada pueden recibirse como un `ParamArray`, algo sobre lo que volveremos en un momento. En el interior de la función simplemente creamos un arreglo con el número de elementos adecuado y copiamos los parámetros en sus elementos, devolviéndolo como resultado.

```
Private Function ArregloEnteros( _
  ByVal ParamArray Elementos()) As Integer()

  Dim Indice As Integer, Arreglo() As Integer

  ReDim Arreglo(Elementos.GetUpperBound(0))

  For Indice = LBound(Elementos) To UBound(Elementos)
    Arreglo(Indice) = Elementos(Indice)
  Next
  Return Arreglo
End Function
```

Disponiendo de la función `ArregloEnteros()`, crear un arreglo en cualquier punto donde se precise sería tan sencillo como escribir dos sentencias. En el fragmento siguiente además se recorre el arreglo para mostrar los valores de sus elementos a efectos de comprobación.

```
Dim MiArreglo() As Integer
Dim Elemento As Integer

MiArreglo = ArregloEnteros(1, 5, 12, 3)

For Each Elemento In MiArreglo
  Console.WriteLine(Elemento)
Next
```

Parámetros por valor y por referencia

Cuando llamamos a un procedimiento y pasamos como parámetro una variable, no una constante, ésta se puede recibir por referencia o por valor. En el primer caso el procedimiento lo que recibe es una referencia a la misma variable que se pasa como parámetro, o lo que es lo mismo, recibe la propia variable. Esto significa que si el procedimiento realiza cualquier modificación la variable original se verá entonces afectada, lo que en algunas ocasiones puede ser útil y en otras un problema. En caso de que una variable se reciba por valor lo que el procedimiento obtiene es una copia del contenido de la variable, pero no la variable original, por lo que cualquier modificación que se efectúe no alterará su valor.

En Visual Basic 6, y versiones previas, por defecto los parámetros se pasaban por referencia. Esto significa que, de no indicarse lo contrario de manera explícita, se asumía el uso de `ByRef`. En Visual Basic .NET, por el contrario, no hay un modo implícito y siempre debe utilizarse `ByVal` o `ByRef`. De no introducirlo nosotros mismos de manera manual, el editor automáticamente añadirá `ByVal` delante de cada parámetro. Esto significa que ahora el modo por defecto es por valor y no por referencia, lo cual tiene ventajas.

Supongamos que hemos escrito una función cuya finalidad es devolver la raíz cuarta del número que se pase como parámetro, como podría ser la siguiente, y que desde un punto del programa se utiliza con una variable que va a ser

posteriormente utilizada para otros fines. Como puede ver, la función `RaizCuarta()` utiliza el propio parámetro, del que recibe una referencia, para realizar una operación intermedia y, dado que la referencia apunta a la variable `Número` original, ésta se habrá visto afectada, por lo que al imprimirla no aparecerá el valor 625, sino 25.

```
Dim Número As Integer
Número = 625
Console.WriteLine("La raíz cuarta es {0},el número original {1}", _
                  RaizCuarta(Número), Número)

Public Function RaizCuarta(ByRef Operando As Integer) As Integer
  Operando = Math.Sqrt(Operando)
  Return Math.Sqrt(Operando)
End Function
```

Nota

La función `Sqrt()` de la clase `Math` devuelve la raíz cuadrada del dato que se entrega como parámetro. Al hallar dos veces la raíz cuadrada de un número lo que se obtiene es la raíz cuarta.

Para evitar este efecto, indeseado en este caso, basta con indicar en el cuerpo de la función, antes de la declaración de cada parámetro, que la variable será recibida por valor. Para ello usaremos `ByVal`, quedando la función como se muestra a continuación. Ahora el valor de la variable `Número` no se verá afectado en ningún caso por la ejecución de `RaizCuarta()`. En realidad, en Visual Basic .NET, basta con no indicar ni `ByVal` ni `ByRef` para usar el primero.

```
Public Function RaizCuarta(ByVal Operando As Integer) As Integer
  Operando = Math.Sqrt(Operando)
  Return Math.Sqrt(Operando)
End Function
```

A diferencia de lo expuesto con este ejemplo, en otros casos el hecho de que el procedimiento que recibe los parámetros pueda modificarlos, todos, algunos o sólo uno, puede ser de interés, ya que, por ejemplo, permitiría devolver más de un valor desde una función. Para indicar que un parámetro se pasa por referencia y no por valor debemos utilizar `ByRef`, según acabamos de ver, en sustitución de `ByVal`.

Parámetros opcionales

Al utilizar un procedimiento o una función es necesario facilitarle tantos parámetros como se hayan especificado en la declaración, en caso contrario obtendremos un error cuando intentemos ejecutarlo. Esto es así porque los parámetros, por defecto, no son opcionales, sino obligatorios. En caso de que uno o

varios de los parámetros sean necesarios unas veces y otras no, lo que haremos será declararlos como opcionales.

Para indicar que un cierto parámetro es opcional pondremos delante de su nombre, y delante de `ByVal` o `ByRef` si es que existen, la palabra `Optional`. Si existe un sólo parámetro opcional éste habrá de situarse en el último lugar, a partir de la declaración de un parámetro opcional todos los siguientes deben ser obligatoriamente opcionales. Es lógico, si tenemos un procedimiento con tres parámetros y resulta que el segundo es opcional y el tercero no, ¿cómo sabrá Visual Basic cuando se faciliten dos parámetros si el segundo corresponde al parámetro opcional o bien al tercero porque el segundo se ha omitido?

En versiones previas de Visual Basic 4.0 cualquier parámetro opcional debía ser de tipo `Variant`, pero a partir de la versión 5 se permite que dichos parámetros tengan tipo como cualquiera otros. Un cambio en Visual Basic .NET es que todos los parámetros opcionales han de tener, obligatoriamente, un valor por defecto. Éste se facilita en la propia lista de parámetros.

La siguiente linea, por ejemplo, definiría un procedimiento que recibe dos parámetros: el primero de tipo `String` y el segundo, opcional, de tipo `Boolean`.

```
Public Sub MuestraCadena(ByVal Cadena As String, _
        Optional ByVal Convertir As Boolean = False)
```

Cuando un procedimiento tiene, como en este caso, parámetros que son opcionales, hay que asumir que al ser invocados es posible que dichos parámetros no sean facilitados. En este caso dichos parámetros tendrán su valor por defecto, `False` en este caso. Continuando con el ejemplo del procedimiento `MuestraCadena()`, supongamos que su finalidad es mostrar la cadena que se pasa como primer parámetro y, en caso de como segundo parámetro se facilite el valor `True` debe convertirla antes en mayúsculas. El procedimiento completo quedaría como se muestra a continuación.

```
Public Sub MuestraCadena(ByVal Cadena As String, _
              Optional ByVal Convertir As Boolean = False)
  If Not Convertir Then
    Console.WriteLine(Cadena)
  Else
    Console.WriteLine(Cadena.ToUpper())
  End If
End Sub
```

Al procedimiento anterior se le podría llamar de distintas formas, siempre que exista al menos un parámetro y éste sea de tipo `String`. Por lo tanto, las siguientes serían llamadas válidas al procedimiento `MuestraCadena()`.

```
Dim Saludo As String = "Hola, ¿qué tal?"

MuestraCadena(Saludo)
MuestraCadena(Saludo, True)
MuestraCadena(Saludo, False)
```

Si necesitamos disponer de un número indeterminado de parámetros opcionales en nuestro procedimiento, en lugar de utilizar `Optional` para definir una serie de variables sin conocer a ciencia cierta cuántas serán necesarias, podemos usar la cláusula `ParamArray`, que nos permite definir como opcional una lista de parámetros en la que podremos recibir cualquier número de parámetros de cualquier tipo. Sólo podemos tener un `ParamArray` en la lista de parámetros de un procedimiento y, en caso de existir, no puede declararse ningún otro parámetro opcional.

Por ejemplo, el siguiente procedimiento puede recibir un número indeterminado de parámetros y además de cualquier tipo.

```
Public Sub MuestraCadenas(ByVal ParamArray Cadena())
  Dim Elemento As Object
  For Each Elemento In Cadena
    Console.WriteLine("{0} -> {1}", _
          Elemento.GetType.ToString(), Elemento)
  Next
End Sub
```

Una llamada válida a este procedimiento podría ser la que se muestra más abajo. Al ejecutar el programa obtendríamos el tipo de cada uno de los parámetros y su contenido, como se aprecia en la figura 6.3.

```
MuestraCadenas(1, "Hola", True, Now, 166.386)
```

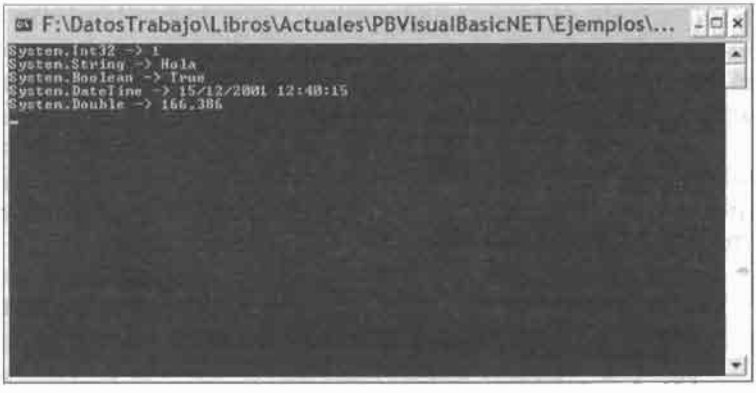


Figura 6.3. Tipos y valores de la lista de parámetros variable

Salida de un procedimiento

Hasta ahora nuestros procedimientos se han ejecutado desde principio a fin, terminando al llegar a la línea `End Sub` que provoca que el control vuelva a la sentencia siguiente a la que provocó la llamada. Al igual que ocurría con los bucles, que en ocasiones era necesario terminarlos antes de tiempo, disponemos de unas sentencias que nos permitirán finalizar la ejecución de un procedimiento de forma anticipada.

Para provocar la salida de un procedimiento usaremos `Exit Sub`, mientras que para realizar la misma operación en una función usaremos `Exit Function`. Generalmente estas sentencias están asociadas a alguna estructura condicional, ya que si las disponemos directamente como una sentencia ejecutable más provocará la salida del procedimiento siempre en el mismo punto, impidiendo así la ejecución del resto del código que exista.

Procedimientos recursivos

Ya hemos visto cómo podemos utilizar un procedimiento y cómo podemos llamarlo desde cualquier punto del módulo, si es privado, o desde cualquier otro, si es global. Estas llamadas siempre son externas, desde fuera del procedimiento, que es el modo más habitual. Sin embargo un procedimiento puede ser llamado desde el interior del código que lo compone, es decir, el procedimiento se puede llamar a sí mismo. Cuando ocurre esto se dice que el procedimiento es recursivo.

Cuando se hace que un procedimiento sea recursivo se está buscando, generalmente, una forma simple de realizar una cierta operación que de otra forma sería más compleja. Hay que tener en cuenta que cada vez que llamamos a un procedimiento, indistintamente del punto desde el que se realice la llamada, en la pila se almacena una dirección de retorno, así como el espacio necesario para las variables locales que tenga el procedimiento.

Por lo tanto, un procedimiento recursivo que se llame a sí mismo un número importante de veces puede provocar un desbordamiento de la pila. Por supuesto esto ocurrirá siempre que el procedimiento se llame continuamente, sin fin, por eso la llamada de un procedimiento a él mismo siempre dependerá de alguna condición.

Como ejemplo suponga que queremos escribir una función a la que facilitándole un número nos devuelva su factorial, es decir, ese mismo número `N` multiplicado por `N-1`, `N-2` y así sucesivamente hasta llegar a multiplicar por 1. Una posible solución sería la función siguiente.

```
Public Function FactorialNoRecursivo(ByVal N As Integer) As Integer
  Dim I, Resultado As Integer
  Resultado = 1
  For I = 2 To N
    Resultado = Resultado * I
  Next
  Return Resultado
End Function
```

Una solución alternativa a ésta es una función recursiva, que se llame tantas veces como sea necesario pasándose en cada llamada el número entregado menos 1. Esto es lo que hace la función `Factorial()`, que si prueba verá cómo obtiene el mismo resultado que la anterior.

```
Public Function Factorial(ByVal N As Integer) As Integer
  If N = 1 Then
```

```
    Return 1
  Else
    Return N * Factorial(N - 1)
  End If
End Function
```

Llamadas a métodos por nombre

Cuando se quiere llamar a un método u obtener o modificar una propiedad de un determinado objeto, lo habitual es usar el sistema que hemos utilizado hasta ahora, consistente en disponer un punto detrás del nombre del objeto seguido del nombre del método o propiedad.

Utilizando este sistema Visual Basic conoce, en el momento de compilar el programa, el objeto y el método o propiedad al que se quiere acceder, por lo que puede establecer un enlace directo.

Existe una alternativa a dicho sistema, mucho más flexible aunque más lenta, que consiste en utilizar la función `CallByName()`. Mediante esta función es posible invocar métodos así como obtener y modificar propiedades sin conocer de antemano cuáles serán, es decir, será al ejecutar el programa cuando se resuelva el enlace y se realice la acción solicitada.

Parámetros al usar `CallByName()`

El número de parámetros que es preciso facilitar a la función `CallByName()` es variable, dependiendo de la acción que se vaya a efectuar y los parámetros que precise el método al que va a llamarse. Los tres primeros parámetros, no obstante, son siempre necesarios.

En primer lugar hay que indicar a qué objeto pertenece el método o la propiedad sobre la que se va a actuar.

Si queremos acceder a una propiedad de un botón llamado `Command1`, por poner un ejemplo, el primer parámetro a entregar a `CallByName()` sería precisamente `Command1`.

Hay que tener en cuenta que lo que facilitamos es un objeto, no el nombre de un objeto, por lo que `Command1` no iría entre comillas.

A continuación tendremos que indicar el nombre del método o propiedad, en este caso facilitando una cadena, que puede ser tanto una constante como una variable u otra propiedad, siempre y cuando el resultado sea de tipo `String`. Esto permite, por ejemplo, solicitar al usuario de un programa el nombre del método que quiere ejecutar.

El tercer parámetro será una de las constantes enumeradas en la tabla 6.1. Con ellas se indica la acción que quiere realizarse, que puede ser una llamada a un método, la obtención del valor actual de una propiedad o su modificación. Estos valores forman parte de una enumeración, llamada `CallType`, definida en el ámbito `Microsoft.VisualBasic`, el mismo en el que se encuentra la función `CallByName()`.

Tabla 6.1. Constantes para el tercer parámetro de `CallByName()`

Constante	Acción a llevar a cabo
`vbMethod`	Ejecución de un método
`vbGet`	Lectura del valor de una propiedad
`vbSet`	Escritura de una propiedad

En caso de que usemos la función `CallByName()` para llamar a un método que no precisa parámetros o para recuperar el valor de una propiedad, será suficiente con los tres parámetros que se han descrito hasta ahora. Si vamos a modificar el valor de una propiedad o llamar a un método que necesita parámetros, sin embargo, deberemos facilitar uno o más parámetros adicionales. El tipo de estos parámetros dependerá del tipo de la propiedad a modificar o de los tipos que precise el método al que se invoca, en realidad `CallByName()` espera como cuarto parámetro un arreglo `Object`.

Puntos clave

- En Visual Basic .NET, como en prácticamente todas las versiones que han existido de BASIC, es posible disponer varias sentencias lógicas en una sola línea, separándolas con dos puntos. Es una técnica, sin embargo, que no contribuye a la claridad del código.
- La sentencia condicional por excelencia es `If`, tras la cual se dispone una expresión que devuelva `True` o `False` o un valor que pueda evaluarse con uno de esos dos.
- Los apartados `Then` y `Else` dan paso a las listas de sentencias que deben ejecutarse en caso de que la expresión sea cierta o falsa, respectivamente.
- Además de la sentencia `If`, las funciones `IIf()`, `Switch()` o `Choose()` pueden resultarnos útiles en casos concretos para verificar una o varias expresiones.
- Cuando sabemos de antemano cuántas veces debe repetirse un bucle utilizamos la sentencia `For Next`, recurriendo en caso contrario a alguna de las variables `Do/Loop`.
- Mediante la sentencia `For Each In` se facilita la enumeración de los elementos de un arreglo o una colección.
- Aunque podemos seguir utilizando la sentencia `On Error`, como en versiones previas de Visual Basic, en Visual Basic .NET contamos con un control estructurado de excepciones, mucho más flexible y claro.
- Mediante los procedimientos y funciones se facilita la reutilización de código. Un procedimiento es un conjunto de sentencias al que se asigna un

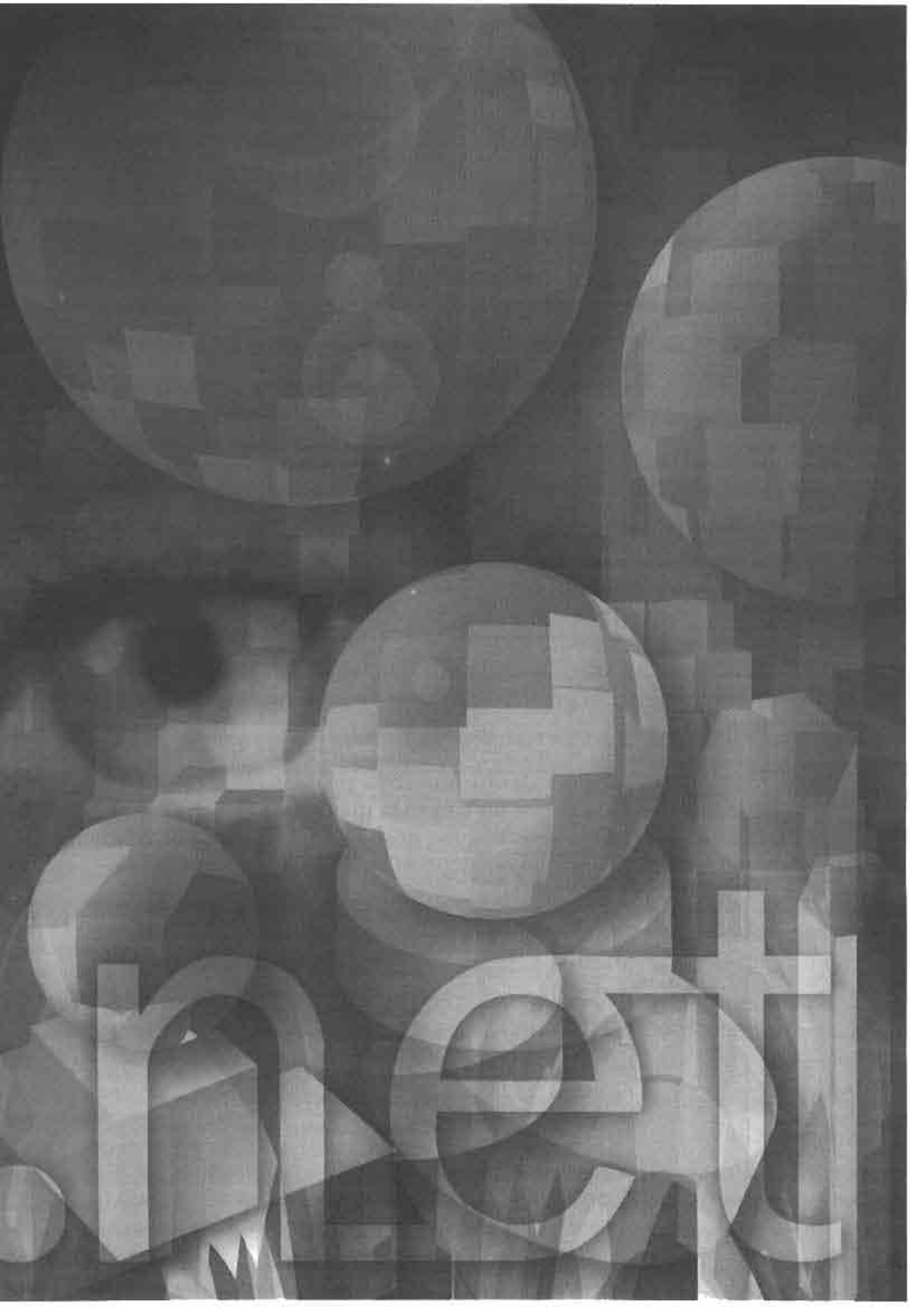
nombre que, posteriormente, puede ser utilizado para ejecutarlas cuando sea necesario.

- Los procedimientos pueden tomar una lista de parámetros. Éstos pueden facilitarse por valor, que es el modo por defecto, o por referencia: `ByVal` y `ByRef`.
- Cuando es necesario devolver un valor se escribe una función en lugar de un procedimiento, utilizando la sentencia `Return` para devolver el resultado.
- Los tipos de los parámetros de entrada y salida pueden ser cualquiera de los que hemos conocido en capítulos previos.
- Es posible invocar a un método, leer o asignar una propiedad mediante la función `CallByName()`.

Resumen

Tal y como ha podido ver en este capítulo, Visual Basic .NET es un lenguaje rico en estructuras de control condicionales, de repetición y estructuración del código en procedimientos y funciones. Todos estos elementos resultarán indispensables en cualquier aplicación que necesitemos desarrollar, independientemente de su finalidad, ya que son fundamentales. El nuevo control estructurado de excepciones es un paso más en la evolución de Visual Basic, equiparándose en este aspecto, la gestión de errores, a los lenguajes de programación más modernos.

En el capítulo siguiente seguiremos avanzando en el conocimiento del lenguaje, aprendiendo a definir nuestras propias clases de objetos y ámbitos, codificando métodos, que no son más que procedimientos pertenecientes a una clase, y utilizando estos objetos desde nuestras propias aplicaciones.



7 Programación orientada a objetos

Visual Basic .NET es la primera versión de Visual Basic verdaderamente orientada a objetos. Aunque en versiones previas era posible crear clases en unos módulos específicos, realmente se trata de un mecanismo dependiente de COM y, además, no se contaba con capacidades fundamentales como la herencia. En Visual Basic .NET, por el contrario, las clases son construcciones de primer nivel equiparables a las de C++, Java o C#.

Nuestro objetivo, en este capítulo, es aprender a definir nuestras propias clases, utilizándolas posteriormente para crear objetos. También estudiaremos algunos temas que guardan cierta relación con la programación orientada a objetos o bien los elementos que puede contener una clase. Algunos de ellos son los ámbitos con nombre, las interfaces o la sobrecarga de métodos.

Ámbitos con nombre

Las aplicaciones actuales son bastante más complejas que las que podían desarrollarse hace unos años sobre DOS o las primeras versiones de Windows. En una aplicación participan ahora un buen número de componentes mediante los cuales se facilita el acceso a bases de datos, la conectividad en redes, comunicación con otras aplicaciones, etc. La existencia de tantos componentes puede

causar conflictos si hay coincidencias de nomenclatura y, por ello, el modelo de componentes más usado hasta ahora, COM (*Component Object Model*), introdujo los conocidos como GUID (*Globally Unique IDentifier*). Estos identificadores son secuencias en 128 bits, o 16 bytes, teóricamente irrepetibles.

Trabajar con GUIDs siempre ha sido farragoso, siendo necesario el uso de archivos de cabecera donde estaban predefinidos o servicios que se encargaban de traducir nombres descriptivos en identificadores únicos, aunque esto no era posible en todos los casos. En la plataforma .NET se ha optado por una solución radicalmente distinta: el uso de espacios o ámbitos con nombre.

Un ámbito con nombre, del inglés *namespace*, es un ámbito delimitado explícitamente al que se ha asignado un identificador. En el interior de un ámbito con nombre es posible incluir tanto definiciones de tipos como otros ámbitos con nombre, creando una anidación que dará lugar a una jerarquía.

En la figura 7.1 se ha representado gráficamente el ámbito con nombre `System` y algunos de los ámbitos que hay en su interior, como `Windows` y `Web`. Éstos contienen, a su vez, otros ámbitos de nombres como `Services` y `UI`.

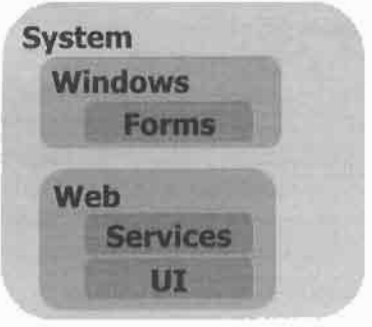


Figura 7.1. Unos ámbitos con nombre contienen otros en su interior

Definición de ámbitos con nombre

Para crear un ámbito con nombre, incluyendo en él las definiciones que nos interesen, utilizaremos la palabra clave `Namespace`. La estructura sería similar a la siguiente:

```
Namespace Nombre

End Namespace
```

Observe que todas las definiciones que se encuentren entre `Namespace` y `End Namespace`, que abren y cierran el bloque, tan sólo existen dentro del espacio con nombre. Esto significa que para hacer referencia a dichas definiciones, por ejemplo para crear un objeto de una cierta clase, habrá que utilizar un nombre cualificado. Es un tema que abordaremos de inmediato.

Como se indicaba antes, un ámbito con nombre puede contener otros en su interior, creando un anidamiento que daría lugar a una cierta jerarquía. Es po-

sible utilizar diferentes sintaxis para incluir un ámbito con nombre en otro. Los dos siguientes fragmentos de código tendrían un resultado equivalente:

```
Namespace Anaya
  Namespace ProgramacionVBNET
     Class HolaMundo
        Public Shared Sub Main()
            System.Console.WriteLine( _
              "Hola desde Anaya.ProgramacionVBNET")
        End Sub
     End Class
  End Namespace
End Namespace

...

Namespace Anaya.ProgramacionVBNET
   Class HolaMundo
      Public Shared Sub Main()
         System.Console.WriteLine( _
          "Hola desde Anaya.ProgramacionVBNET")
      End Sub
   End Class
End Namespace
```

En ambos casos tenemos un ámbito con nombre, llamado `Anaya`, que contiene en su interior otro, denominado `ProgramacionVBNET`. Este, finalmente, contiene la clase `HolaMundo`. El resultado de ejecutar cualquiera de estos dos programas es exactamente el mismo: la salida de una cadena de caracteres por la consola. La clase `HolaMundo` no está en un ámbito global, sino en un espacio con nombre llamado `Anaya.ProgramacionVBNET`, de forma similar a como la clase `Form` se encuentra en el espacio con nombre `System.Windows.Forms`.

Nota

Para seguir los ejemplos inicie un nuevo proyecto Visual Basic .NET de tipo consola, eligiendo el elemento Aplicación de consola, y sustituya el código generado por el asistente por el código del ejemplo. A continuación abra la ventana de propiedades y seleccione la clase en la lista Objeto inicial, como se ha hecho en la figura 7.2. Siga este mismo sistema para los ejemplos siguientes, ya que no vamos a usar un módulo, como en capítulos previos, sino clases que contienen un método compartido `Main()`.

A la hora de crear nuevos ámbitos con nombre deberemos tratar siempre de crear referencias inequívocas. En los fragmentos anteriores, por ejemplo, se ha usado el nombre de una empresa seguida del nombre de un libro, de tal forma que difícilmente puede encontrarse la referencia `Anaya.ProgramacionVBNET` en otra parte. En realidad, `Anaya.ProgramacionVBNET` sería un subámbito que existiría en el ámbito de primer nivel, establecido en la ventana mostrada en la figura 7.2. Por defecto, Visual Basic genera un ámbito con el nombre del

proyecto, aunque la definición no es visible en forma de ninguna sentencia `Namespace`. Nuestra clase, en este caso concreto, se encontraría en el ámbito `HolaMundoConsola.Anaya.ProgramacionVBNET`.

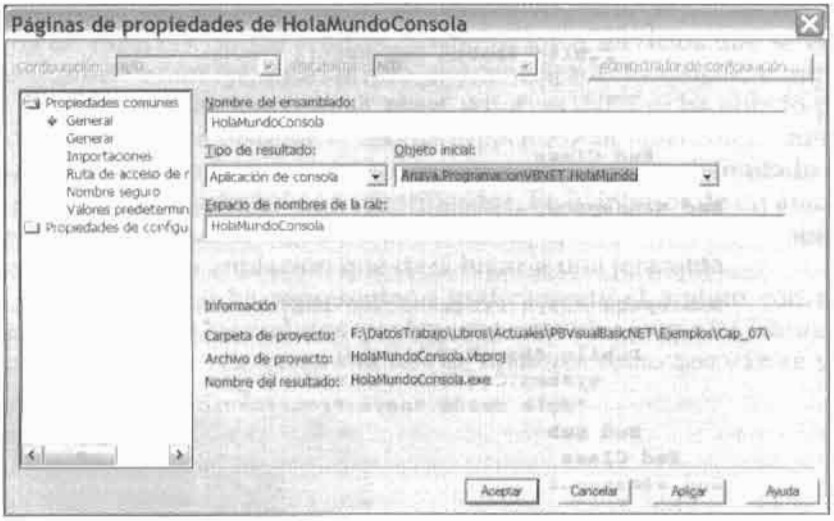


Figura 7.2. Establecemos el punto de entrada a la aplicación

Referencias a un ámbito con nombre

Suponiendo que la clase `HolaMundo` que hemos definido fuese pública, que en este momento no lo es, y deseásemos crear un objeto de dicha clase desde otro programa, tendríamos que componer una referencia completa que permitiese al compilador identificarla de manera inequívoca. Esto significaría anteponer al nombre de la clase el del ámbito o ámbitos con nombre donde se encuentra definida, como se hace en este programa:

```
Class UsaHolaMundo
  Shared Sub Main()
    Dim MiHolaMundo As Anaya.ProgramacionVBNET.HolaMundo =
      New Anaya.ProgramacionVBNET.HolaMundo()

    System.Console.WriteLine(MiHolaMundo.Saludo())
  End Sub
End Class
```

En este caso `Anaya.ProgramacionVBNET` es, al igual que `System`, el identificador de un ámbito con nombre. `HolaMundo`, por el contrario, es el identificador de una clase, como lo es `Console`.

Una alternativa a esta composición de referencias cualificadas consiste en utilizar la sentencia `Imports`, seguida del ámbito cuya referencia no deseamos repetir continuamente. *El resultado, como puede ver a continuación, es un código más legible.*

```
Imports System
Imports Anaya.ProgramacionVBNET

Class UsaHolaMundo
  Shared Sub Main()
    Dim MiHolaMundo As HolaMundo = New HolaMundo()

    Console.WriteLine(MiHolaMundo.Saludo())
  End Sub
End Class
```

Observe que mediante `Imports` tan sólo puede hacerse referencia a ámbitos con nombre, no a clases, estructuras o cualquier otra definición de tipo. No es válido, por tanto, intentar lo siguiente. Si compilásemos este código obtendríamos el correspondiente error.

```
Imports System.Console
...
   WriteLine("Hola")
```

Conflictos entre ámbitos con nombre

Utilizar la palabra clave `Imports` para hacer visible en nuestro ámbito elementos contenidos en un cierto espacio con nombre, que es lo que hemos hecho en el punto anterior, puede dar origen a conflictos y ambigüedades. Para verlo, lo mejor es servirnos de ejemplos reales con código.

Vamos a partir de un nuevo módulo Visual Basic .NET con el código mostrado a continuación. En él tenemos, como puede verse, un ámbito con nombre que tiene en su interior a otros dos. Cada uno de estos define una clase llamada `HolaMundo`. Podemos compilar este código, generando una biblioteca de clases (véase figura 7.3), sin que se produzca error alguno. Esto es así porque aunque existen dos clases con el mismo nombre, `HolaMundo`, cada una se encuentra en un ámbito distinto.

```
Namespace ClasesAnaya


    Namespace ProgramacionVSNET

        Public Class HolaMundo

            Public Function Saludo() As String
                Return _
                  "Hola desde el ámbito Programacion.GuiaVSNET"
            End Function
        End Class
    End Namespace

    Namespace ProgramacionVBNET

        Public Class HolaMundo
```

```
            Public Function Saludo() As String
                Return _
                  "Hola desde el ambito Anaya.ProgramacionVBNET"
            End Function
        End Class
    End Namespace

End Namespace
```

Figura 7.3. Iniciamos un proyecto Biblioteca de clases para alojar nuestro código

Nota

Para evitar que el ámbito `ClasesAnaya` esté, a su vez, contenido en un ámbito superior definido automáticamente por Visual Basic .NET, abra la ventana de propiedades del proyecto y elimine el contenido del apartado Espacio de nombres de la raíz.

Utilizar las clases definidas en el módulo anterior es bastante sencillo, básicamente basta con añadir una referencia al proyecto en que se necesiten y, por supuesto, incluir las respectivas referencias mediante la instrucción `Imports`. Inicie un nuevo proyecto añadiéndolo a la solución actual, en lugar de cerrarla, de tipo aplicación de consola. Abra el menú emergente del proyecto, en el Explorador de soluciones, y seleccione la opción Agregar referencia. En la ventana que aparece, similar a la de la figura 7.4, elija de la página **Proyectos** la única opción disponible. Finalmente, introduzca este código en el módulo de la aplicación de consola.

```
Imports System
Imports ClasesAnaya.ProgramacionVSNET
Imports ClasesAnaya.ProgramacionVBNET
```

```
Module Module1

    Sub Main()

        Dim MiHolaMundo As HolaMundo = New HolaMundo()

        Console.WriteLine(MiHolaMundo.Saludo())
    End Sub

End Module
```

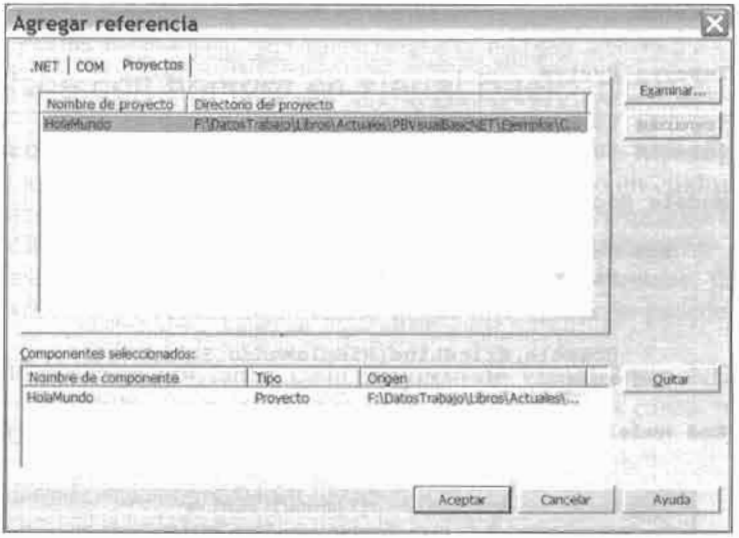


Figura 7.4. Añadimos en el proyecto de consola una referencia a la libreria de clases

Al intentar compilar este código, como se ha hecho en la figura 7.5, verá que obtenemos un error. En el método `Main()` usamos la clase `HolaMundo`, pero resulta que existen dos clases con ese nombre, una en el ámbito de `ClasesAnaya.ProgramacionVSNET` y otra en el de `ClasesAnaya.ProgramacionVBNET`. La solución a este problema pasa, lógicamente, por utilizar referencias completas como las mostradas en un ejemplo anterior de este mismo capitulo.

Nota

En realidad no es necesario llegar a compilar para ver el problema, puesto que Visual Basic .NET va efectuando una compilación en segundo plano y señalando los puntos donde hay errores a medida que editamos. Basta con situar el puntero del ratón sobre la marca, tal como se ha hecho en la figura 7.5, para obtener una indicación de cuál es el problema.

Ya vemos que la solución es obvia, pero no lo es menos la incomodidad de tener que introducir referencias como:

```
Dim MiHolaMundo As ClasesAnaya.ProgramacionVSNET.HolaMundo = _
     New ClasesAnaya.ProgramacionVSNET.HolaMundo();
```

para poder crear un objeto de una cierta clase. Afortunadamente, la sentencia `Imports` nos permite importar el contenido de un ámbito con nombre creando un alias, un pseudónimo al que podemos asignar el identificador que nos convenga. Esto es lo que se hace en el ejemplo siguiente. En él puede observar cómo se asignan los alias `VS` y `VB` a los dos ámbitos con nombre, de tal manera que, posteriormente, podemos utilizar dichos alias en lugar de las referencias completas.

```
Imports System

Imports VS = ClasesAnaya.ProgramacionVSNET
Imports VB = ClasesAnaya.ProgramacionVBNET

Module Module1

    Sub Main()
        Dim MiHolaMundo As VB.HolaMundo = _
              New VB.HolaMundo()

        Console.WriteLine(MiHolaMundo.Saludo())
    End Sub

End Module
```

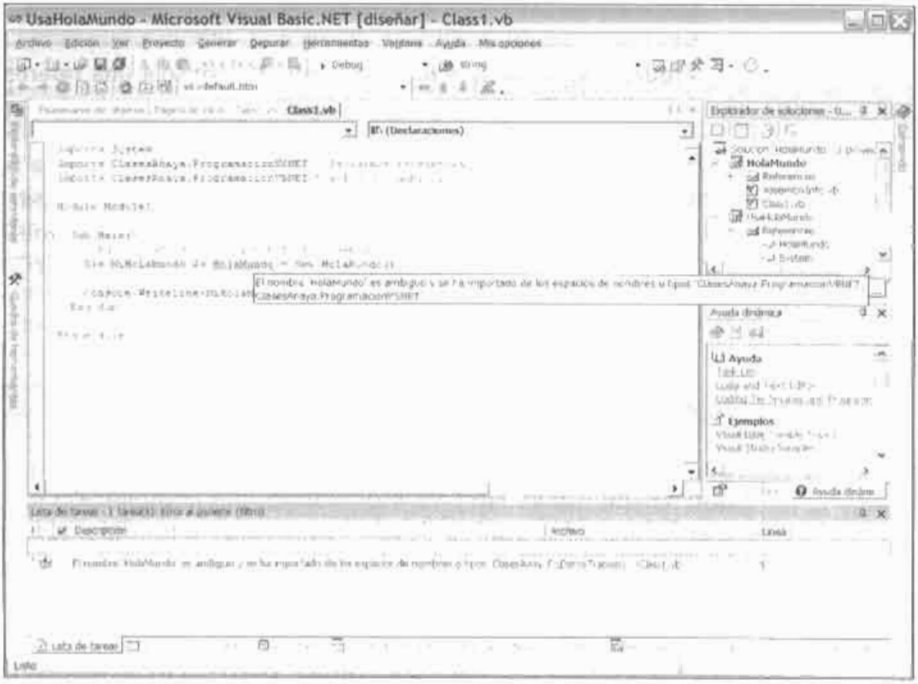


Figura 7.5. En la Lista de tareas aparecen las indicaciones de error

Nota

A la hora de crear identificadores y construir referencias debe tener en cuenta que el lenguaje Visual Basic .NET, a diferencia de otros existentes en Visual Studio .NET, no distingue entre mayúsculas y minúsculas. Esto significa que si define un ámbito con nombre, una clase o cualquier otro tipo con el nombre `Vbasic` por poner un ejemplo, y utiliza posteriormente el nombre `VBasic` estaría haciendo referencia al mismo elemento, no a dos distintos.

Ámbitos con nombre en Visual Basic .NET

Como acabamos de ver, utilizando la palabra clave `Namespace` podemos definir los ámbitos con nombre que necesitemos, como en cualquier otro de los lenguajes .NET. Con Visual Basic .NET, sin embargo, hemos de tener en cuenta que cada vez que se inicia un nuevo proyecto automáticamente se crea un ámbito para contener sus elementos. Ese ámbito, como ya sabe, tiene el nombre del proyecto y podemos modificarlo desde la ventana de propiedades del proyecto.

En la práctica, por tanto, cada proyecto de Visual Basic .NET ya tiene implícito un ámbito con nombre que, en la mayoría de los casos, nos será útil tal cual, sin necesidad de crear otros de manera explícita.

Clases de objetos

En un capítulo previo pudimos conocer los tipos de datos intrínsecos de Visual Basic .NET, así como la forma de crear nuevos tipos que almacenan valor: enumeraciones y estructuras. Las clases representan otro mecanismo para crear nuevos tipos de datos, si bien en este caso las variables contendrían la referencia a los objetos creados a partir de la clase. Las clases, por tanto, se definen, igual que definimos una enumeración o una estructura, describiendo los miembros con que contará. Posteriormente, usando la clase como si fuese un tipo, se crearían los objetos y accedería a sus miembros.

Lo que diferencia a las clases de cualquier otro tipo de dato, aparte de que se utilicen para crear objetos de los cuales se obtiene una referencia, es que pueden derivarse unas de otras, posibilidad inexistente en las estructuras, por poner un ejemplo. No tenemos por qué crear una clase partiendo de cero, aunque podemos hacerlo. Podríamos crear una clase nueva de formulario partiendo de la clase `Form` ya existente, heredando todas sus características.

Al igual que las estructuras, las clases pueden implementar interfaces, tema que abordaremos en este mismo capítulo, así como contar entre sus miembros con variables de datos, procedimientos, funciones y propiedades. Además también pueden incluirse delegados y eventos. Algunos de estos elementos los conocerá no en éste sino en el próximo capítulo.

Los puntos siguientes introducen las características más importantes relativas a la definición de clases en Visual Basic .NET, acompañando las explicaciones con algunos ejemplos breves. Para obtener una descripción completamente detallada de cada modificador o técnica consulte la referencia del lenguaje Visual Basic .NET.

Definición de una clase

Ya en los ejemplos previos ha visto, de forma básica, cuál es la definición de una clase sencilla. La palabra clave es `Class`, tras la cual dispondremos el identificador de la clase. La lista de miembros iría en las líneas siguientes, hasta finalizar la definición con `End Class`. Tanto la clase como sus miembros pueden contar con múltiples modificadores, parte de los cuales va a tener ocasión de conocer de inmediato.

Las clases se definen en el interior de un ámbito con nombre. A diferencia de éstos, que son meros contenedores, las clases son estructuras de implementación. Las clases se usan para modelar las aplicaciones como si estuvieran compuestas de objetos más o menos discretos. Al igual que ocurre con los ámbitos, en el interior de una clase es posible definir otras.

De manera implícita, a menos que indiquemos lo contrario, toda clase está derivada del tipo `Object` que, como ya sabe, es la base de todos los tipos de datos de Visual Basic .NET. Éste cuenta con una serie de miembros que, por tanto, son heredados por todas las clases que podamos definir. Si nuestra clase está derivada de otra heredará, asimismo, todos los miembros de su clase base y los de las clase base de la base, ascendiendo en la jerarquía hasta llegar a `Object`.

Visibilidad de una clase y sus miembros

Las clases, como acaba de decirse, pueden estar incluidas en un ámbito con nombre o en otras clases. Su visibilidad, fuera de ese ámbito, dependerá de los modificadores que dispongamos delante de la palabra `Class` en el momento de la definición. Estos modificadores pueden también aplicarse a los miembros de la clase: métodos, propiedades, variables, etc. Las posibilidades son las siguientes:

- `Public` – La clase, o el miembro, es visible en todos los ámbitos, no sólo en el que se ha definido. Las clases públicas, por ejemplo, pueden utilizarse desde ámbitos con nombre externos al propio. Los miembros públicos de una clase son accesibles, al crear un objeto, desde fuera de la propia clase.
- `Protected` – Este modificador sólo es aplicable a los miembros de una clase, no a la clase en sí. La visibilidad está reducida a la propia clase y las que pudieran derivarse de ella. Es decir, un miembro protegido no es accesible externamente.

- `Private` – Es el ámbito más reducido. Una clase privada sólo puede utilizarse en el interior del ámbito en que se encuentra incluida, ya sea un *namespace* u otra clase. Los miembros que tienen este modificador, igualmente, sólo pueden ser usados desde el interior de la clase donde se han definido, nunca desde fuera, ni siquiera en clases derivadas.
- `Friend` – Es similar a `Public`. Si bien la clase o el miembro es visible desde fuera de su ámbito inmediato, esta regla sólo se aplica dentro del ensamblado al que pertenece. Una clase `Public` puede utilizarse desde otros ensamblados, mientras que una `Friend` no.
- `Protected Friend` – Como puede suponer, es una combinación de `Protected` y `Friend`. El resultado es que los identificadores pueden utilizarse en el ensamblado en que se han definido, así como en clases derivadas a pesar de que se encuentren en otros ensamblados.

La visibilidad de las clases y sus miembros es algo que debe deducirse directamente del diseño lógico de la propia aplicación. Normalmente ésta constará de una serie de clases públicas a partir de las cuales se crearán los objetos fundamentales, exponiendo métodos y propiedades que permitan a las aplicaciones configurar su funcionamiento. Internamente, estas clases pueden utilizar otras no visibles desde el exterior, así como miembros privados y protegidos.

Clases derivadas y herencia

Uno de los mecanismos más interesantes de los lenguajes orientados a objetos, categoría en la que cae Visual Basic .NET, es el de herencia. Gracias a él, es posible definir una nueva clase partiendo de otra existente, de tal forma que la derivada hereda de la base todos sus miembros, lo cual implica contar, ya de partida, con la misma funcionalidad. La clase derivada puede acceder a todos los miembros de la base exceptuando aquellos que son privados.

Mediante el mecanismo de herencia se contribuye a la reutilización del código, algo habitual en todos los lenguajes orientados a objetos. La novedad, en Visual Basic .NET, es que dicho código no tiene por qué estar escrito en el mismo lenguaje. Desde Visual Basic .NET podemos derivar una nueva clase tomando como base otra escrita en C#, J#, C++ o COBOL, por poner algunos ejemplos. Tenemos, de esta forma, lo mejor del modelo de componentes COM, que era la independencia de lenguaje, y lo mejor de los lenguajes orientados a objetos, la herencia de implementación inexistente en COM.

Como se dijo anteriormente, todas las clases derivan por defecto de `Object`. Si deseamos otra base distinta tendremos que indicarla mediante la palabra clave `Inherits`, como puede verse en el ejemplo mostrado a continuación.

```
Public Class ClaseBase

  Public Function Saludo() As String
    Return "Hola mundo"
```

```
  End Function
End Class

Public Class ClaseDerivada
  Inherits ClaseBase

  Public Function Cuadrado(ByVal N As Integer) As Integer
    Return N * N
  End Function
End Class
```

La clase `ClaseDerivada` hereda los métodos de `ClaseBase` y también los de `Object`, que es la ascendiente de `ClaseBase`. Disponiendo el código anterior en un módulo de un proyecto de consola, podríamos añadir el fragmento siguiente para crear un objeto de la clase `ClaseDerivada` e invocar a los métodos `Cuadrado()` y `Saludo()`.

```
Public Class Aplicacion
  Public Shared Sub Main()

    Dim MiClase As ClaseDerivada = New ClaseDerivada()

    Console.WriteLine("{0}, {1}", _
       MiClase.Saludo(), MiClase.Cuadrado(5))
  End Sub
End Class
```

En caso de que haya escrito una clase de la cual no le interese que se deriven otras, por considerarla una clase de uso final, no tiene más que anteponer la palabra `NotInheritable` delante de `Class`. Pruebe a ponerla en la clase `ClaseBase`, verá como no puede compilar el programa obteniendo un mensaje de error (véase figura 7.6). Esa clase, por lo tanto, sólo podría utilizarse para crear objetos, pero nunca para derivar otras partiendo de ella.

También podemos encontrarnos justamente en el caso contrario: haber definido una clase cuyo objetivo es servir de base para otras. La clase puede no tener funcionalidad propia, sirviendo simplemente como raíz común de clases que compartirían algunos miembros.

Suponga que va a definir una serie de clases que representarán entidades gráficas y, para facilitar el trabajo, quiere que todas esas clases compartan una serie de miembros comunes. Definir una raíz común tendrá diversas ventajas. Esta clase base podría ser la siguiente:

```
Public MustInherit Class Entidad
  Protected X, Y As Integer
  Protected Color As Integer

  Public Sub Dibuja()

  End Sub
End Class
```

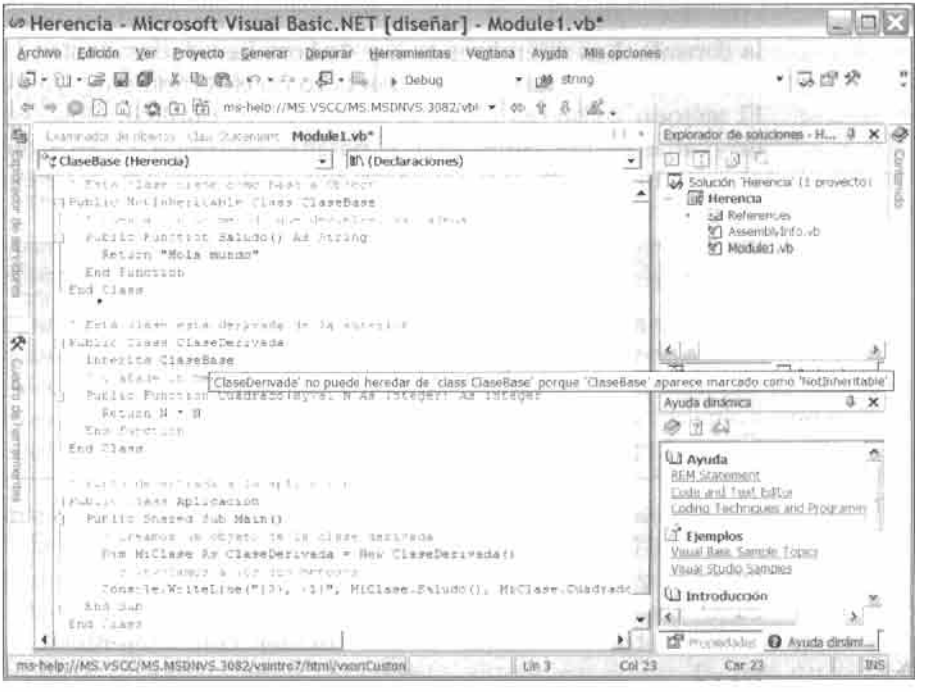


Figura 7.6. Mensaje que obtenemos al intentar derivar de una clase `NotInheritable`

Observe que delante de la palabra `Class` hemos dispuesto el modificador `MustInherit`. Con él indicamos que no es posible crear objetos directamente con la clase `Entidad`, siendo necesario derivar otras específicas. Dos de ellas podrían ser como las mostradas a continuación:

```
Public NotInheritable Class Punto
  Inherits Entidad
  Public Shadows Sub Dibuja()
    Console.WriteLine( _
     "Un punto de color {0} en la posición {1},{2}", Color, X, Y)
  End Sub
End Class

Public NotInheritable Class Linea
  Inherits Entidad
  Private X1, Y1 As Integer

  Public Shadows Sub Dibuja()
    Console.WriteLine( _
       "Una linea de color {0} desde {1},{2} a {3},{4}", _
       Color, X, Y, X1, Y1)
  End Sub
End Class
```

Estas dos clases derivan de `Entidad` y, además, son `NotInheritable`, es decir, son clases de uso final, no pudiendo derivarse otras nuevas a partir de

ellas. En realidad no llegan a dibujar nada, limitándose a mostrar los datos por la consola. Los miembros `X`, `Y` y `Color`, heredados de `Entidad`, pueden utilizarse desde las clases derivadas pero no externamente, ya que son protegidos. El método `Dibuja()`, por el contrario, es público. Observe que en las clases derivadas hemos antepuesto la palabra `Shadows` a `Sub`. Con ella se indica que el método que está definiéndose oculta al heredado de la clase base.

Nota

Las clases que sirven como raíz pero no pueden ser usadas para crear objetos, como es el caso de `Entidad`, se denominan *clases abstractas*.

Constructores y destructores

Al igual que en C++, Java, Object Pascal o C#, las clases definidas en Visual Basic .NET pueden contar con unos métodos específicos para controlar la construcción y destrucción de objetos. Esos métodos son conocidos genéricamente como constructores y destructores. Una misma clase puede contar con múltiples constructores, siempre que cuenten con diferentes listas de parámetros, mientras que destructor sólo puede haber uno por clase. Los constructores siempre tienen por nombre `New()` y no cuentan con valor de retorno, pudiendo tomar o no parámetros según interese. El destructor no tiene valor de retorno y no puede tomar parámetro alguno. Su nombre será `Finalize()`.

Nota

En cierta forma, el constructor y destructor de una clase Visual Basic .NET equivalen a los constructores `Class_Initialize` y `Class_Terminate` de un módulo de clase de Visual Basic 6.0.

En el momento en que se crea un objeto de una cierta clase, utilizando para ello el operador `New`, se ejecuta automáticamente el constructor que corresponda según los parámetros facilitados. El destructor de una clase se ejecuta para un cierto objeto automáticamente cuando ya no hay ninguna referencia a él, es decir, no necesitamos destruir los objetos explícitamente, es un trabajo del que se ocupa el sistema de recogida de basura de la plataforma .NET. Debido a esto no es posible asegurar que el destructor para un cierto objeto se ejecute de manera inmediata en cuanto éste ya no está en uso, puesto que la recogida de objetos no útiles se efectúa en momentos concretos en los cuales no hay otra actividad pendiente.

El siguiente fragmento muestra una clase con un constructor sin parámetros, un constructor con un parámetro y el destructor. No es funcional puesto que no se hace nada en el interior de los métodos y tampoco se ha codificado un punto de entrada en la aplicación.

```
Class Agenda

    Public Sub New()
    End Sub

    Public Sub New(ByVal Parametro As String)
    End Sub

    Protected Overrides Sub Finalize()
    End Sub
End Class
```

Para crear objetos de esta clase podríamos, por lo tanto, utilizar el operador `New` sin entregar parámetro alguno o bien facilitando una cadena, como se hace en las dos sentencias siguientes:

```
Dim MiAgenda As Agenda = New Agenda()

Dim MiAgenda As Agenda = New Agenda("Parámetro")
```

Nota

Observe que el destructor no es un procedimiento público sino protegido, puesto que no es accesible desde el exterior de la clase y se ejecuta automáticamente. Además, dado que es un método que heredamos en la clase `Agenda` de la clase `Object`, para redefinirlo es necesario utilizar la palabra clave `Overrides`.

Las clases se definen, por regla general, para poder crear objetos a partir de ellas, aunque también es cierto que una clase puede contar con miembros compartidos o estáticos y ser utilizada directamente, sin crear objeto alguno. Mediante el uso de miembros compartidos es posible mantener información común entre todos los objetos de una clase. Dichos miembros no pertenecen a un objeto en particular, sino a la clase, por lo que son creados la primera vez que es utilizada y no cada vez que se crea un objeto.

Está claro, por tanto, que los constructores clásicos no pueden utilizarse para ejecutar el proceso de inicio que establezca la clase en su estado por defecto, generalmente asignando unos valores determinados a los miembros estáticos. Si utilizásemos un constructor para esto, realmente el proceso de iniciación se ejecutaría al crear cada nuevo objeto, con lo que la información compartida se perdería constantemente.

En Visual Basic .NET existe el concepto de *constructor de clase*, un tipo de constructor que no puede tomar parámetros y cuyo único modificador posible, obligatorio además, es `Shared`. Este constructor se ejecuta tan sólo una vez: cuando se aloja la clase en memoria. Es el punto adecuado, por tanto, para preparar los miembros compartidos.

Clases anidadas

Como se indicó anteriormente, en el interior de una clase es posible definir otras clases, creando un efecto de anidación. Esto es útil cuando una cierta clase tan sólo va a ser necesaria o útil en el interior de otra, nunca fuera. Suponga que tiene una clase `Agenda` que va a gestionar anotaciones en forma de objetos `Anotacion`. Esta última clase, suponiendo que nunca va a ser utilizada con otro fin, podría definirse en el interior de `Agenda`.

Anidar una clase en otra es también una forma de ocultarla ya que, no haciéndola pública, no sería posible usarla desde el exterior ni aún componiendo una referencia completa. Suponiendo que tenemos una clase `Agenda` y en su interior otra llamada `Anotacion`, una sentencia como la siguiente generaría un error a menos que hubiésemos usado el modificador `Public` al definir la segunda clase.

```
MiAnotacion As Agenda.Anotacion = _
     New Agenda.Anotacion()
```

Conseguimos, por tanto, la encapsulación de una clase en el interior de otra, aislándonos del exterior. El código siguiente es un programa completo en el que puede verse la definición de las clases `Agenda` y `Anotacion`, con sus constructores y destructores, así como el uso desde otra clase que forma parte de la misma aplicación.

```
Class Agenda

    Class Anotacion

        Shared Sub New()
            Console.WriteLine("Constructor de clase de Anotacion")
        End Sub

        Public Sub New()
            Console.WriteLine("Constructor de Anotacion")
        End Sub

        Public Sub New(ByVal Parametro As String)
            Console.WriteLine("Constructor {0}", Parametro)
        End Sub

        Protected Overrides Sub Finalize()
            Console.WriteLine("Destructor de Anotacion")
        End Sub
    End Class

    Private MiAnotacion As Anotacion

    Public Sub New()
```

```
        Console.WriteLine("Constructor Agenda")

        MiAnotacion = New Anotacion("Nueva anotación")
    End Sub

    Protected Overrides Sub Finalize()
        Console.WriteLine("Destructor Agenda")

        MiAnotacion = Nothing
    End Sub
End Class

    Class Principal
        Shared Sub Main()
            Dim MiAgenda As Agenda = New Agenda()
            Console.WriteLine("Objeto creado")
            MiAgenda = Nothing
            Console.WriteLine("Fin de la aplicación")
        End Sub
    End Class
```

Al ejecutar este programa podrá ver el orden en que se ejecutan los diferentes constructores y el destructor de la aplicación. Tal y como se aprecia en la figura 7.7, al ejecutarse el destructor se genera una excepción. Esto ocurre porque la aplicación termina y, posteriormente, el recolector de basura destruye el objeto y se ejecuta el destructor. En ese momento el flujo abierto por la aplicación para usar la consola ha sido cerrado, y el destructor, al enviar un mensaje, genera la excepción.

Figura 7.7. Resultado obtenido al ejecutar el ejemplo de clases anidadas

Objetos de una clase

La finalidad última de las clases, a excepción de aquellas que son abstractas, es facilitar la creación de objetos que gestionen un cierto proceso u operación

necesaria en un programa. Los objetos se crean a partir de las clases usando el operador `New`, según ha podido ver en algunos de los ejemplos previos. En ese momento se asigna un bloque de memoria para el objeto y se inicia su estado, dando el valor por defecto a sus miembros y, si procede, ejecutando el constructor apropiado. Lo que obtenemos en la variable es una referencia a ese objeto, es decir, la dirección del objeto en memoria conjuntamente con información de tipo.

Cada objeto de una clase contará con su propia copia de los miembros de datos, mientras que compartirá con los demás objetos de la misma clase los métodos y propiedades. Cuando se invoca a uno de esos métodos se usa la notación `Objeto.Metodo()`, de tal forma que se facilita la referencia del objeto sobre el que debe actuarse. El método utiliza esa referencia para acceder a los datos propios de ese objeto y no de otro.

Cuando codificamos un método de una clase, por lo tanto, al acceder a los miembros de datos estamos utilizando implícitamente la referencia al objeto que corresponda. Si queremos usar una referencia explícita debemos utilizar el identificador `Me`. Éste representa al objeto que ha llamado al método.

Miembros de una clase

Una clase es como un molde cuya configuración la establecen sus miembros. Éstos pueden ser variables: para contener información relativa a la clase o a cada objeto creado a partir de ella; propiedades: miembros que permiten personalizar los objetos u obtener información de ellos; métodos: procedimientos y funciones que operan sobre esos datos, y eventos: señales que permiten a los objetos interactuar con la aplicación en que son usados.

También es posible introducir en una clase definiciones de enumeraciones, estructuras y constantes.

Independientemente de su categoría, todos los miembros de una clase pueden utilizar los modificadores de visibilidad `Private`, `Protected`, `Public`, `Friend` y `Protected Friend` comentados con anterioridad. Ellos determinarán los puntos en los que cada miembro será o no accesible. Además de la visibilidad, hay otros factores a tener en consideración durante la definición de los miembros de una clase.

Miembros compartidos

Una clase puede constar básicamente con dos categorías de miembros: aquellos que son asociados a cada objeto creado a partir de la clase, que suelen denominarse miembros de *instancia*, y los que, por el contrario, están asociados a la clase en sí, no a un objeto en particular. Estos últimos son miembros compartidos por todos los objetos de la clase y se distinguen por contar con el modificador `Shared`. En otros lenguajes de programación se denominan miembros estáticos.

Partamos de una clase sencilla, como es la siguiente, en la que existen dos miembros: una variable privada y un método público que la manipula y muestra por la consola:

```
Class Contador
  Private N As Integer
  Public Sub Incrementa()
    N += 1
    Console.WriteLine(N)
  End Sub
End Class
```

Cada vez que creemos un objeto `Contador` éste contará con su propia copia de la variable `N`, de tal manera que el método `Incrementa()` actuará sobre una instancia particular de dicha variable, sin afectar a las de otros objetos. Podemos verlo con una simple prueba:

```
Dim Objeto1 As Contador = New Contador()
Dim Objeto2 As Contador = New Contador()
Objeto1.Incrementa()
Objeto2.Incrementa()
Objeto1.Incrementa()
Objeto2.Incrementa()
```

Al ejecutar este código obtenemos la salida mostrada en la figura 7.8. Está claro que `Objeto1` opera sobre una variable `N` que en nada afecta a la homónima de `Objeto2`, como debe ser. Proyectando este comportamiento en otras clases, como pueda ser un botón en una ventana, encontramos el sentido. Al modificar el título de un botón, por ejemplo, no se afectará en ningún momento a los títulos de los demás botones que pudieran existir en la ventana. Los botones son objetos creados a partir de una clase en la que el miembro que aloja el título no es compartido.

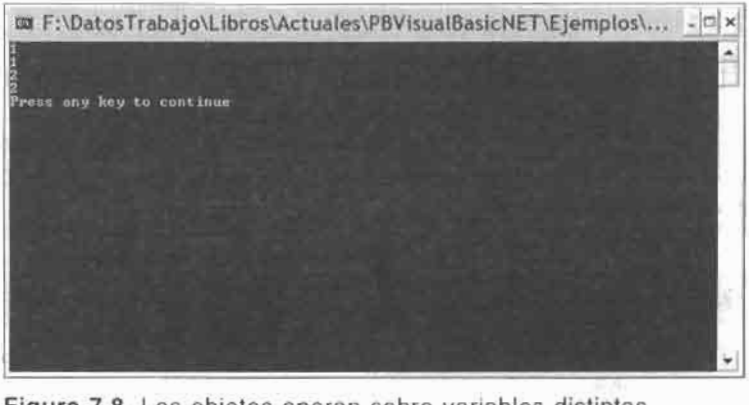


Figura 7.8. Los objetos operan sobre variables distintas

Si modificamos la declaración del miembro `N` de la clase `Contador`, poniendo la palabra `Shared` detrás de `Private`, comprobaremos entonces que el

comportamiento es bien distinto. Al ejecutar el programa obtenemos una secuencia en la que se aprecia que `Objeto1` y `Objeto2` incrementan y muestran la misma variable. Es un mecanismo, por tanto, que permite compartir información entre todos los objetos de una misma clase.

No sólo las variables pueden ser compartidas, también es una posibilidad aplicable a los métodos. Un método compartido puede invocarse simplemente disponiendo delante el nombre de la clase, sin necesidad de crear previamente un objeto. El método `WriteLine()` de la clase `Console`, por poner un ejemplo, es compartido, por ello podemos llamarlo sin crear un objeto de la clase `Console`.

Un método compartido no está asociado a ninguna instancia de clase, es decir, a ningún objeto creado a partir de ella. Esto implica que al ser invocado no reciba la referencia al objeto sobre el que debe actuar, tal y como se describía anteriormente.

No es posible, por tanto, usar el identificador `Me` para operar sobre el objeto implícito porque éste no existe.

Pruebe a añadir el siguiente método a la clase `Contador` definida anteriormente.

```
Public Shared Sub Reinicia()
  N = 0
End Sub
```

Es un método compartido y, como tal, no podría acceder a la variable `N` a menos que ésta también fuese compartida. Asumiendo que esto es así, con el código siguiente reiniciamos el contador sin necesidad de crear objeto alguno, usando el propio nombre de la clase.

```
Objeto1.Incrementa()
Objeto2.Incrementa()
Contador.Reinicia()
Objeto1.Incrementa()
Objeto2.Incrementa()
```

Los métodos compartidos, también conocidos como *métodos de clase*, son apropiados cuando facilitan una operación en la que no está implicado un objeto concreto y, por lo tanto, la referencia implícita no es necesaria. Ya se ha puesto como ejemplo el método `WriteLine()` de la clase `Console`. Otros los encontramos en los métodos de la clase `Math`.

Nota

Tal y como se señalaba en el punto dedicado a la construcción y destrucción de objetos, una clase puede contar también con un constructor de clase o constructor compartido. Éste no puede tomar parámetros y se ejecuta en cuanto se inicia el programa, habitualmente estableciendo valores iniciales en los miembros compartidos.

Miembros sobrecargados

Otra de las nuevas posibilidades de Visual Basic .NET es la *sobrecarga* de métodos, característica que nos permite tener en una misma clase varios métodos con el mismo nombre siempre que la lista de parámetros sea distinta. De esta forma no tenemos porque *inventar* nombres distintos para denominar a métodos que, realizando la misma función, toman parámetros distintos.

Para indicar que un determinado método sobrecarga a otro se utiliza la palabra clave `Overloads`. En realidad, no es necesario indicarlo de manera explícita siempre que los distintos métodos formen parte de la misma clase. Si el método sobrecargado se hereda de una clase base, por el contrario, sí es obligatorio utilizar `Overloads` ya que, de lo contrario, el compilador no sabría si lo que se quiere es sobrecargar, redefinir u ocultar el método original.

En multitud de ejemplos, de este capítulo y los anteriores, hemos utilizado el método `WriteLine()` para mostrar datos por la consola. En ocasiones hemos entregado como único parámetro una cadena o una variable entera, mientras que en otros los parámetros han sido varios. Esto es posible gracias a la sobrecarga y, gracias a ella, no hay necesidad de tener un método `WriteString()`, otro `WriteBoolean()`, otro `WriteInteger()`, etc.

La clase siguiente es un ejemplo sencillo en el que se ofrece un método con dos versiones distintas: uno para dibujar puntos y otro para circunferencias. Los dos métodos son compartidos, de tal forma que no es necesario crear un objeto para invocarlos.

```
Public Class Lienzo

  Public Shared Sub Dibuja(ByVal X As Integer, ByVal Y As Integer)
    Console.WriteLine("Punto en {0},{1}", X, Y)
  End Sub


  Public Shared Sub Dibuja(ByVal X As Integer, _
                           ByVal Y As Integer, ByVal R As Integer)
    Console.WriteLine( _
      "Circunferencia con centro en {0},{1} y radio {2}", X, Y, R)
  End Sub
End Class
```

Desde el método `Main()` de nuestro programa podríamos usar las dos sentencias siguientes para llamar a esos dos métodos:

```
Lienzo.Dibuja(10, 20)
Lienzo.Dibuja(100, 100, 50)
```

Aunque en este caso podría resultar lógico contar con un método `Punto()` y otro `Circunferencia()`, en la práctica podríamos tener múltiples métodos `Circunferencia()`, cada uno de ellos aceptando una lista de parámetros distinta: un centro y un radio, dos esquinas de un cuadrado donde se incluiría, etc.

Observe que no hemos utilizado la palabra `Overloads` ya que, al estar los dos métodos en la misma clase, Visual Basic asume que deseamos sobrecargarlos. Un caso distinto sería el de la clase siguiente:

```
Public Class LienzoExtendido
  Inherits Lienzo
  Public Overloads Shared Sub Dibuja( _
    ByVal X As Integer, ByVal Y As Integer, ByVal C As String)
    Console.WriteLine("Punto en {0},{1} con color {2}", X, Y, C)
  End Sub
End Class
```

En este caso `Overloads` resulta imprescindible porque, de lo contrario, obtendríamos un error al no saber si se quiere sobrecargar nuevamente el método, contando con tres versiones distintas de él, o quiere redefinirse u ocultarse.

Nota

La sobrecarga es aplicable también a los constructores, de tal forma que podemos tener varios aceptando distintos parámetros para establecer la configuración inicial del objeto.

Miembros redefinidos y ocultos

La derivación de una nueva clase partiendo de otra existente, como ya sabe, implica la herencia de todos sus miembros, entre otros los métodos con que pudiera contar la que actúa como base. Es lo que ha ocurrido, por ejemplo, al definir `LienzoExtendido`, clase que puede utilizarse para llamar a cualquiera de las tres versiones del método sobrecargado.

Es relativamente fácil encontrarse en una situación en la que un método heredado no implementa su función tal y como es necesaria en la clase derivada, siendo necesario redefinirlo u ocultarlo. La sobrecarga en este caso no es aplicable, ya que no nos interesa contar con dos versiones del mismo método y, además, la nueva versión probablemente tendrá la misma lista de parámetro que la definida en la clase base, lo cual haría inviable el uso de `Overloads`.

Suponga que ha definido una clase, como `Lienzo`, con métodos para dibujar puntos, circunferencias, líneas, etc. Esta clase contaría con un método, por ejemplo llamado `Dibuja()`, que sería el encargado de llevar todas esas entidades al dispositivo final de dibujo: la pantalla. Posteriormente decide derivar una clase tomando a `Lienzo` como base. Su finalidad será permitir exactamente las mismas operaciones de dibujo pero, en lugar de enviarlas a la pantalla, enviándolas a la impresora. El método `Dibuja()`, por tanto, tendría que ser redefinido. El caso sería el siguiente:

```
Public Class Lienzo
```

```
    Public Overridable Sub Dibuja()
      Console.WriteLine("Dibujando en pantalla")
    End Sub
  End Class

  Public Class LienzoImpresora
    Inherits Lienzo
      Método redefinido para dibujar en impresora
    Public Overrides Sub Dibuja()
      Console.WriteLine("Dibujando en la impresora")
    End Sub
  End Class
```

Observe que el método de la clase base ha sido definido como `Overridable`, permitiendo así su redefinición en las clases derivadas. Éstas deben usar el modificador `Overrides` para indicar que redefinen el método, es decir, facilitan una nueva implementación para ese miembro, no una nueva versión. Al llamar al método `Dibuja()` de un objeto de la clase `Lienzo` se ejecutará la implementación de esa clase, mientras que en un objeto de la clase `LienzoImpresora` se ejecutaría el de dicha clase.

Para redefinir un método es necesario que su *firma*, compuesta por el nombre, lista de parámetros y tipo de retorno, sea idéntica a la del método redefinido. Una alternativa es la creación de un nuevo método que oculta al de la clase base, en lugar de redefinirlo. Ese nuevo método debe tener el mismo nombre, pero por lo demás puede diferir en parámetros y tipos. Es más, incluso puede ser otro tipo de miembro, como una propiedad. Para ocultar un miembro heredado tan sólo hay que cambiar la palabra `Overrides` por `Shadows`. Las implicaciones de usar uno u otro método las conocerá de inmediato, al abordar el uso polimórfico de objetos.

Que un método cuente con el modificador `Overridable` indica que se permite su redefinición en las clases derivadas, pero no se exige. Otros dos modificadores, relacionados con éste, son `NotOverridable` y `MustOverride`. Como puede suponer, el primero indica que un método no puede ser redefinido, mientras que el segundo cambia ese *permite* por una *obligación* de redefinir el método en las clases derivadas.

Nota

El uso conjunto de `MustInherit`, en una clase, y `MustOverride`, en sus miembros, facilita la creación de clases base que actúan como plantillas de partida para crear otras más especializadas.

Acceso a los miembros de la clase base

Cuando en una clase derivada se oculta o redefine un método, en ocasiones puede ser necesario invocar, desde el nuevo método, al que se había heredado

de la clase base. Es algo habitual cuando la nueva clase lo que hace es complementar en cierta medida el funcionamiento de la que actúa como base, pero necesitando aún la implementación de ésta. Suponga que la clase `Lienzo` efectúe un proceso de preparación para almacenar toda la información de las entidades, proceso que en `LienzoImpresora` debe complementarse con la configuración de la propia impresora. Desde el constructor de `LienzoImpresora`, por tanto, habría que invocar al de `Lienzo`.

En estos casos, al necesitar desde un método de una clase acceder a los de la clase base, usaremos el identificador `MyBase`. Recuerde que `Me` representa a la instancia, al objeto, de una clase determinada. `MyBase`, por su parte, representa a la porción de ese objeto implementada por la clase base.

Podemos complementar las clases `Lienzo` y `LienzoImpresora` precisamente con ese funcionamiento, implementando un constructor en la clase base y otro en la derivada, invocando al primero desde el segundo.

```
Public Class Lienzo

  Public Sub New()
    Console.WriteLine("Preparamos el almacenamiento de entidades")
  End Sub



  Public Overridable Sub Dibuja()
    Console.WriteLine("Dibujando en pantalla")
  End Sub
End Class

Public Class LienzoImpresora
  Inherits Lienzo

  Public Sub New()

    MyBase.New()
    Console.WriteLine("Preparamos la impresora")
  End Sub

  Public Overrides Sub Dibuja()
    Console.WriteLine("Dibujando en la impresora")
  End Sub
End Class
```

El resultado de ejecutar el fragmento de código siguiente, en el que se crea un objeto de la clase `LienzoImpresora` y se invoca a su método `Dibuja()`, es el mostrado en la figura 7.9. En él puede ver cómo se ha ejecutado primero el constructor de la clase base, después el de la derivada y, finalmente, el mencionado método `Dibuja()`.

```
Dim Objeto As LienzoImpresora = New LienzoImpresora()
Objeto.Dibuja()
```

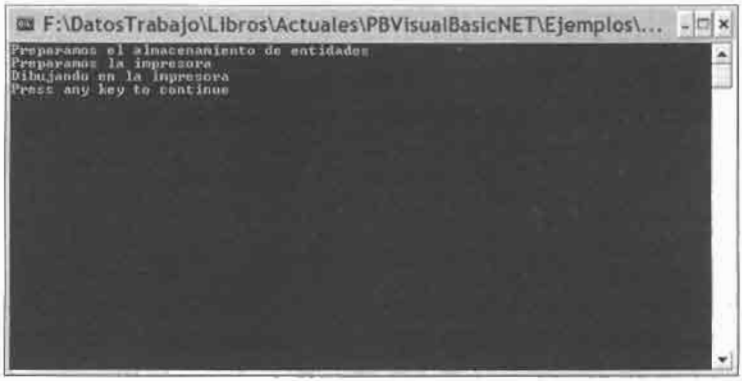


Figura 7.9. Sucesión de llamadas al crear el objeto `LienzoImpresora`

Uso polimórfico de los objetos

La herencia de clases no sólo representa una de las formas de reutilización del código sino que, además, habilita un mecanismo conocido como *polimorfismo*, gracias al cual es posible utilizar objetos de distintas clases, pero que comparten una raíz común, de manera genérica. En definitiva, el polimorfismo nos permite ahorrar trabajo, así como extender la funcionalidad de las clases de objetos sin, por ello, tener que conocer de antemano todos los tipos derivados.

Para comprender el concepto de polimorfismo hay que entender antes que cualquier objeto creado a partir de una clase derivada de otra es, en cierta forma, un superconjunto de la base. Suponga que ha construido un circuito digital con las funciones de un reloj y, tomando éste como base, ha añadido los elementos necesarios para realizar cálculos simples. Cuenta con dos circuitos distintos, uno para montar en carcasas de reloj de pulsera y otro para colocarlo en cajas de calculadora con reloj. El segundo circuito, en caso necesario, podría montarse en una caja de reloj porque cuenta con esas funciones. El primero, por el contrario, no podría utilizarse en una calculadora por razones obvias: no dispone de la lógica para realizar cálculos.

Algo similar ocurre con las clases que están derivadas unas de otras. Si tenemos una clase `Rectangulo` derivada de `EntidadGrafica`, una variable de este segundo tipo puede contener una referencia a un objeto de la primera ya que, al fin y al cabo, un objeto `Rectangulo` contiene toda la funcionalidad de un objeto `EntidadGrafica` más elementos específicos y propios. Puede verlo en el código siguiente que define las dos clases mencionadas y otras dos derivadas de `EntidadGrafica`.

```
Public MustInherit Class EntidadGrafica

  Protected iX1, iY1, iX2, iY2 As Integer

  Protected sColor As String
```

```
  Public Sub New(ByVal X1 As Integer, _
  ByVal Y1 As Integer, ByVal X2 As Integer, _
  ByVal Y2 As Integer, ByVal Color As String)
    iX1 = X1
    iX2 = X2
    iY1 = Y1
    iY2 = Y2
    sColor = Color
  End Sub

  Public MustOverride Sub Dibuja()
End Class

Public Class Rectangulo
  Inherits EntidadGrafica

  Public Sub New(ByVal X1 As Integer, _
  ByVal Y1 As Integer, ByVal X2 As Integer, _
  ByVal Y2 As Integer, ByVal Color As String)
    MyBase.New(X1, Y1, X2, Y2, Color)
  End Sub

  Public Overrides Sub Dibuja()
    Console.WriteLine("Rectángulo ({0},{1})-({2},{3}) color {4}", _
    ix1, iy1, ix2, iy2, scolor)
  End Sub
End Class

Public Class Ovalo
  Inherits EntidadGrafica

  Public Sub New(ByVal X1 As Integer, _
  ByVal Y1 As Integer, ByVal X2 As Integer, _
  ByVal Y2 As Integer, ByVal Color As String)
    MyBase.New(X1, Y1, X2, Y2, Color)
  End Sub

  Public Overrides Sub Dibuja()
    Console.WriteLine("Óvalo ({0},{1})-({2},{3}) en color {4}", _
    ix1, iy1, ix2, iy2, scolor)
  End Sub
End Class

Public Class Linea
  Inherits EntidadGrafica

  Public Sub New(ByVal X1 As Integer, _
  ByVal Y1 As Integer, ByVal X2 As Integer, _
  ByVal Y2 As Integer, ByVal Color As String)
```

```
    MyBase.New(X1, Y1, X2, Y2, Color)
  End Sub

  Public Overrides Sub Dibuja()
    Console.WriteLine("Linea ({0},{1})-({2},{3}) en color {4}", _
    ix1, iy1, ix2, iy2, scolor)
  End Sub
End Class
```

Nota

En la definición de las clases derivadas se usa `Overrides` para redefinir un método de la clase base. Ese método heredado podría ser invocado, usando `MyBase`, en caso de no haber utilizado `MustOverride` en la clase base, igual que hacemos en los constructores de las clases derivadas para llamar al constructor de la base.

Suponga que desea crear una composición gráfica formada por una línea, un rectángulo y un óvalo. En principio podría pensar en crear un objeto de cada una de esas clases asignando las respectivas referencias a una variable específica, procediendo a continuación a llamar al método `Dibuja()` de cada uno de ellos. Es lo que se hace en el siguiente fragmento de código:

```
Dim UnaLinea As Linea = New Linea(0, 0, 320, 240, "Rojo")
Dim UnRectangulo As Rectangulo = _
   New Rectangulo(10, 10, 100, 100, "Negro")
Dim UnOvalo As Ovalo = New Ovalo(50, 70, 80, 11, "Verde")

UnaLinea.Dibuja()
UnRectangulo.Dibuja()
UnOvalo.Dibuja()
```

En este caso estamos tratando a cada elemento como un objeto de una clase distinta y, a pesar de que lo son, no tenemos en cuenta que todos ellos comparten una raíz común. Aprovechando esta característica, sería más fácil usar el código siguiente para conseguir el mismo objetivo, especialmente si en lugar de tres entidades estamos trabajando con decenas de ellas.

```
Dim Composicion() As EntidadGrafica = _
  {New Rectangulo(10, 10, 100, 100, "Negro"), _
   New Ovalo(50, 70, 80, 110, "Verde"), _
   New Linea(0, 0, 320, 240, "Rojo")}

Dim Elemento As EntidadGrafica

For Each Elemento In Composicion

  Elemento.Dibuja()
Next
```

En este caso no tenemos varias variables independientes sino un arreglo de tipo `EntidadGrafica`. Dado que esta clase es base de `Rectangulo`, `Ovalo` y `Linea`, los elementos del arreglo pueden ser de cualquiera de esas clases. Al recorrer el arreglo llamando al método `Dibuja()` obtenemos el resultado de la figura 7.10. A pesar de que el tipo de `Elemento` es `EntidadGrafica`, al llamar al método `Dibuja()` se utiliza la implementación específica de la clase de objeto que corresponda. Esto es, precisamente, el polimorfismo y, gracias a él, podemos tratar de forma genérica objetos de distintas clases.

Figura 7.10. Cada objeto invoca al método correspondiente a su tipo

Otro caso típico en el que es interesante este tratamiento genérico, por mencionar los más usuales, se da cuando implementamos alguna función o procedimiento cuya finalidad es operar sobre objetos que tienen una característica común. Suponga que va a implementar un procedimiento al cual facilitándole un objeto de una clase derivada de `EntidadGrafica` se encargue de escribir sus datos en un archivo, para lo cual usaría un método que en la clase base habría declarado como `MustOverride`, obligando a su implementación en cualquier derivada. Ese hipotético método no tiene por qué conocer clases como `Linea`, `Ovalo` o cualquier otra clase que pudiéramos haber definido, usando `EntidadGrafica` como tipo del parámetro a recibir:

```
Sub Guarda(ByVal Entidad As EntidadGrafica)
  Console.WriteLine(Entidad.Escribe())
End Sub
```

Siempre que el objeto facilitado a `Guarda()` esté derivado de `EntidadGrafica`, no importa su tipo, este procedimiento podrá efectuar su trabajo sin problemas.

Interfaces

A diferencia de C++, en el que es posible derivar una clase de múltiples bases, en Visual Basic .NET no existe el concepto de herencia múltiple, como en

muchos otros lenguajes. Al definir una clase no podemos, por tanto, especificar dos clases base. Sí es posible, no obstante, indicar que se implementan múltiples interfaces.

Una interfaz es una especie de contrato en el que se enumeran los métodos que la componen, indicando tipos de retornos y listas de parámetros. En lenguajes como C y C++ las interfaces no existen como construcción nativa del lenguaje, por lo que se definen en forma de estructuras o bien clases abstractas con métodos virtuales puros. En contraposición, Visual Basic .NET sí dispone de interfaces como elementos nativos del lenguaje.

Nota

Las interfaces habilitan asimismo el uso polimórfico de objetos de diversas clases sin la exigencia de que todas ellas tengan una raíz común, bastando la implementación en todas ellas de la misma interfaz.

Definición de una interfaz

Salvo por el hecho de que los miembros de una interfaz no cuentan con implementación, lo cierto es que su sintaxis es prácticamente idéntica a la de una clase cualquiera, pudiendo contar con métodos, propiedades, indexadores, etc. Básicamente se trata de sustituir la palabra `Class` por `Interface`, teniendo en cuenta, por supuesto, que lo que estamos definiendo es un acuerdo o especificación al que podrán ajustarse posteriormente las clases que lo necesiten, no una implementación.

Supongamos que vamos a crear varias clases que actuarán como agenda, por lo que deseamos que, de cara al usuario de esas clases, todas compartan una interfaz común. De esta forma, el programador que las utilice podrá elegir entre una clase u otra, por ejemplo dependiendo de dónde desee almacenar la información, pero sabiendo siempre que contará con los mismos mecanismos de acceso a la clase.

El siguiente fragmento de código define una interfaz, llamada `IAgenda`, declarando un método `Enumera()`. En este caso es el único miembro, pero podríamos añadir lo que deseásemos. Fíjese en que no existe una implementación.

```
Interface IAgenda
  Sub Enumera()
End Interface
```

Implementación de una interfaz

La clase que necesita implementar una determinada interfaz, por ejemplo una hipotética clase `Agenda`, tan sólo tiene que indicarla tras la palabra `Implements`. Esta formará parte de la cabecera de la clase, como la palabra clave `Inherits`. La diferencia es que al derivar de una clase se heredan todos sus

miembros, incluida la implementación de los métodos, mientras que al indicar la implementación de una interfaz lo único que obtenemos es la obligación de implementar los miembros definidos en ella.

Una clase tan sólo puede derivar de otra, como se indicaba antes, pero nada le impide implementar varias interfaces. En este caso se indicarían los nombres de esas interfaces uno tras otro separados por comas. Los métodos correspondientes a cada interfaz deberán indicarlo con la misma palabra `Implements`.

El siguiente ejemplo muestra dos interfaces diferentes, así como una clase que implementa una de ellas y otra que implementa las dos. En realidad no hay código ejecutable que haga nada útil, es simplemente una demostración de la sintaxis a utilizar.

```
Module Module1

  Sub Main()
    ' El tipo del array es IAgenda, una interfaz
    Dim MiAgenda() As IAgenda = {New Agenda(), New AgendaArchivo()}
    ' Invocamos al método que corresponda en cada objeto
    MiAgenda(0).Enumera()
    MiAgenda(1).Enumera()
  End Sub

End Module

' La interfaz IAgenda [illegible]
Interface IAgenda
  Sub Enumera()
End Interface

' La interfaz IPersistente [illegible]
Interface IPersistente
  Function Almacena() As Boolean
  Function Recupera() As Boolean
End Interface

' La clase Agenda, la correspondiente a la interfaz IAgenda,
' [illegible]
Class Agenda
  Implements IAgenda

  Public Function Anade( _
    ByVal Anotacion As String) As Boolean
    Return True
  End Function

  Public Sub Enumera() Implements IAgenda.Enumera
    Console.WriteLine("Clase Agenda")
  End Sub
End Class

' [illegible]
Class AgendaArchivo
  Implements IAgenda, IPersistente

  Public Sub Enumera() Implements IAgenda.Enumera
```

```
    Console.WriteLine("Clase AgendaArchivo")
  End Sub

  Public Function Almacena() As Boolean _
     Implements IPersistente.Almacena
    Return True
  End Function

  Public Function Recupera() As Boolean _
     Implements IPersistente.Recupera
    Return True
  End Function
End Class
```

Observe que los métodos que implementan la funcionalidad definida en las interfaces lo indican usando la misma palabra `Implements` seguida del nombre de la interfaz y el propio método. En el método `Main()` usamos un arreglo de tipo `IAgenda` para tratar de manera polimórfica objetos de dos clases que, sin tener una base específica como ocurría con `EntidadGrafica`, implementan una misma interfaz.

Módulos

Además de clases y definiciones de tipos, tales como estructuras y enumeraciones, en un ámbito podemos encontrar también *módulos*. Un módulo es una clase con algunas características implícitas: no es posible derivarlo de otra clase ni usarlo como base para la definición de una nueva, todos sus métodos son compartidos, a pesar de no usar explícitamente el modificador `Shared`, y su visibilidad está abierta a todo el módulo de código en el que está definido.

El módulo, como ha podido ver en algunos de los ejemplos propuestos, se usa habitualmente para introducir el método `Main()` que actúa como punto de entrada a la aplicación. Nada nos impide, sin embargo, usar el módulo para alojar otros métodos e, incluso, tener varios módulos lógicos en el mismo módulo físico de código, aunque no sea habitual.

Puntos clave

- Las definiciones de tipos e identificadores se alojan en Visual Basic .NET en el interior de ámbitos con nombre, espacios lógicos a los que se asigna un nombre.
- Los tipos definidos en el interior de un ámbito con nombre forman un grupo lógico que puede ser importado desde otros ámbitos en caso necesario.
- Al importar un ámbito con nombre en otro es posible asociar un alias al nombre original del ámbito para evitar conflictos entre identificadores.

- Visual Basic .NET es un lenguaje totalmente orientado a objetos y dispone de las características esenciales de este tipo de lenguajes: encapsulación, herencia y polimorfismo.
- Las clases son tipos de datos a partir de los cuales podemos crear objetos. Éstos pueden contener miembros de datos, métodos, propiedades, delegados y eventos.
- Cada clase puede tener uno o más constructores, todos ellos llamados `New()`, y un destructor que será ejecutado automáticamente por el recolector de basura.
- Visual Basic .NET contempla la sobrecarga de métodos, de forma que es posible tener varias funciones o procedimientos que comparten un nombre pero toman una lista distinta de parámetros.
- Gracias a la herencia es posible crear una clase a partir de otra ya existente, heredando todos sus miembros que, en caso de necesidad, pueden ser redefinidos y ocultados.
- Cada vez que se crea un objeto éste dispone de una copia de los datos y comparte la funcionalidad de los métodos definidos en la clase.
- Una clase puede contar con miembros compartidos, llamados estáticos en otros lenguajes de programación, habilitando así el uso compartido de datos entre objetos.
- Mediante el polimorfismo es posible tratar de manera genérica objetos que, siendo de distintas clases, cuentan con una raíz común o bien implementan una misma interfaz.
- Las interfaces son definiciones sin implementación, pensadas para aplicarse en la codificación de clases que deben contar con unos servicios bien conocidos.
- Los módulos son un caso particular de clase con algunas características específicas.

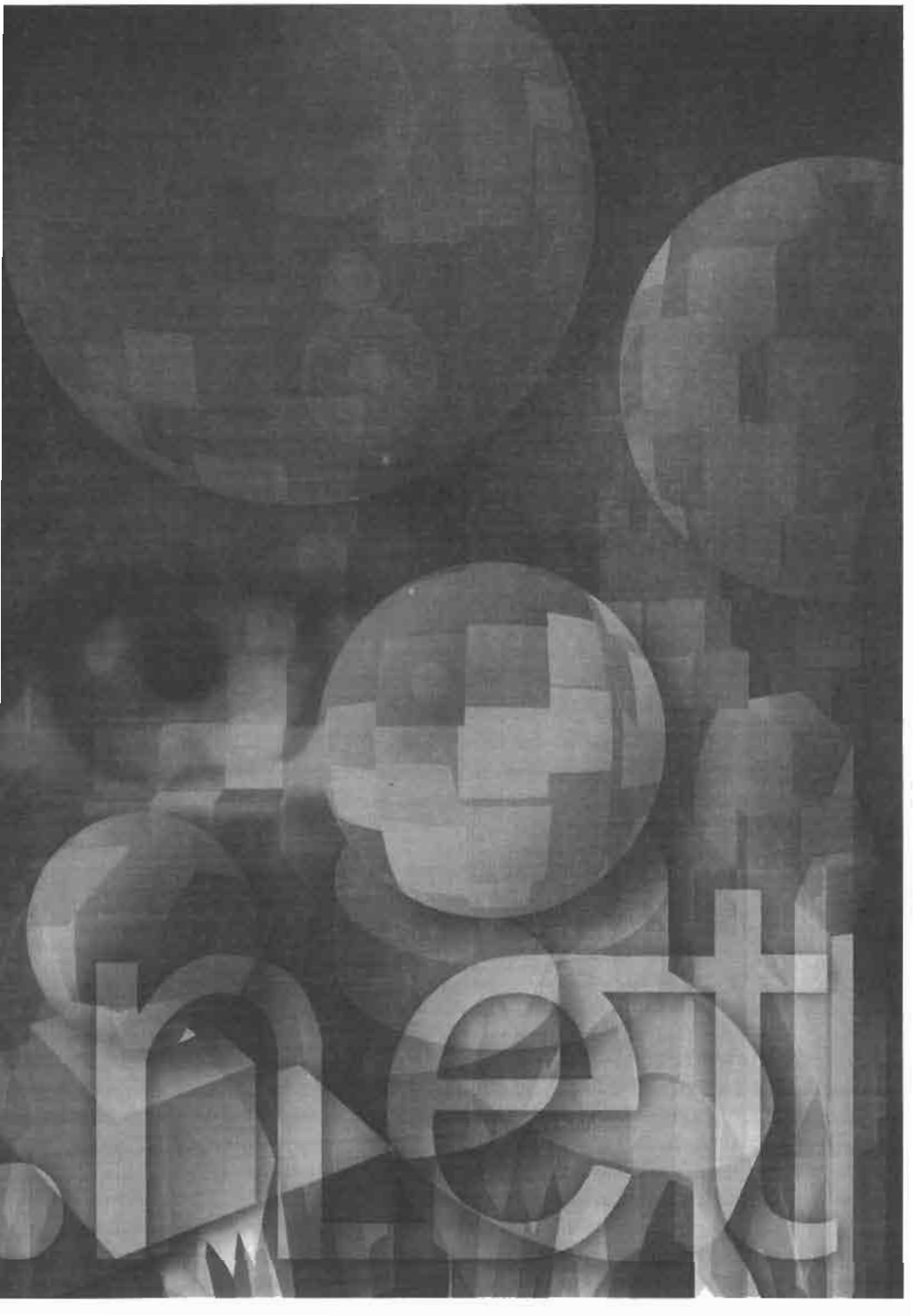
Resumen

Como ha podido ver en este capítulo, Visual Basic .NET incorpora novedades en el lenguaje que elevan su categoría hasta el mismo nivel que otros lenguajes orientados a objetos, tales como C++, Java o C#. Respecto a las versiones previas de Visual Basic, contamos con la posibilidad de definir clases sin usar módulos específicos, los conocidos módulos de clase de Visual Basic 6, aparte de contar con posibilidades tan importantes como la herencia, sobrecarga de métodos o la definición e implementación de interfaces.

En el interior de una clase, como se ha explicado, es posible introducir definiciones de nuevos tipos, como enumeraciones u otras clases, así como miem-

bros de datos y métodos. De esta manera se encapsula en los objetos una funcionalidad discreta que, posteriormente, puede ser utilizada a demanda en las aplicaciones donde sea necesario. Son las ventajas de un lenguaje orientado a objetos.

Visual Basic .NET no es sólo un lenguaje orientado a objetos, sino también orientado al uso de componentes, como en versiones previas. El próximo capítulo le guiará en el proceso de añadir propiedades, usar delegados y codificar eventos en sus clases, convirtiéndolas en verdaderos componentes reutilizables de manera visual.



8

Programación orientada a componentes

Uno de los factores determinantes para el éxito de Visual Basic, desde su aparición, ha sido la facilidad de utilizar componentes prefabricados para desarrollar aplicaciones propias. Primero fueron los conocidos como VBX, después los OCX o componentes ActiveX y ahora, con Visual Basic .NET, tenemos los componentes .NET.

La programación de aplicaciones usando componentes fue lo que dio origen al término RAD (*Rapid Application Development*), tan difundido posteriormente para otros lenguajes y sistemas.

Los componentes son objetos creados a partir de una clase, si bien el objetivo es que dichos objetos puedan crearse de manera visual, por ejemplo introduciéndolos en un contenedor mediante un diseñador, y personalizarse mediante el establecimiento de propiedades y asociación de código a sus eventos. Una clase de componente, por tanto, no sólo contará con miembros de datos y métodos, disponiendo también esos elementos.

Nuestro objetivo, en este capítulo, es aprender a añadir propiedades, delegados y eventos a nuestras clases, dando así el primer paso para su conversión en clases de componentes.

En capítulos posteriores profundizaremos sobre las particularidades de los componentes .NET así como los puntos específicos para los controles Windows y Web.

Propiedades

Las clases almacenan información que, en muchas ocasiones, no sólo es para uso interno sino que también puede interesar al programador que las utilice. Ya sabemos que los datos se almacenan en campos de datos o variables, cuya visibilidad podemos controlar mediante diversos modificadores. Suponga que desea que el valor de una cierta variable pueda ser leído externamente pero no modificado, o bien que cada vez que se modifique pueda controlar el nuevo valor para saber si es válido. Éstos son algunos de los casos adecuados para definir una propiedad.

Al igual que las variables, las propiedades tienen un tipo que indica la naturaleza de la información que contienen. A diferencia de ellas, sin embargo, la asignación o lectura no se efectúa de manera directa sino a través de la ejecución de métodos de lectura y escritura.

Definición de una propiedad

Una propiedad aparece a los ojos del usuario de una clase como si fuese una variable, pero en realidad internamente existen unos métodos de acceso que son los que controlan la lectura o asignación de valores. La definición se efectúa con la palabra `Property`, tras la cual dispondremos el nombre de la propiedad, la palabra `As` y el tipo. Opcionalmente, delante de `Property` es posible usar los modificadores de ámbito que conocimos en el capítulo previo. La siguiente línea, por ejemplo, sería el encabezado de una propiedad de tipo `String` llamada `Nombre`.

```
Public Property Nombre() As String
```

El final de la definición vendrá marcada por `End Property`. En el cuerpo de la propiedad introduciremos dos apartados, llamados `Get` y `Set`, que serán ejecutados cuando se recupere el valor de la propiedad y se asigne, respectivamente. Se asume que el apartado `Get` devolverá un valor del tipo indicado en la cabecera de la propiedad, mientras que `Set` tomará dicho parámetro. Lo habitual es que el valor se almacene internamente en alguna variable y se devuelva cuando sea necesario, aunque en realidad nada nos impide implementar una funcionalidad distinta.

A continuación puede ver la definición completa de la hipotética propiedad `Nombre()`. En este fragmento de código se asume que `Pnombre` es una variable privada declarada al principio de la clase y usada para almacenar el valor de la propiedad.

```
Public Property Nombre() As String
  Get
    Return Pnombre
  End Get
  Set(ByVal Value As String)
```

```
        Pnombre = Value
      End Set
    End Property
```

Aunque en este ejemplo nos limitamos a almacenar el valor o devolverlo, en la práctica podríamos efectuar comprobaciones o elaborar el valor a devolver a partir de otros datos. El acceso a la propiedad, creando un objeto de esta clase, sería idéntico al acceso a cualquier miembro de datos.

Propiedades de sólo lectura y sólo escritura

Mientras no se indique lo contrario de manera explícita, toda propiedad debe contar con los apartados `Get` y `Set`, de tal manera que el usuario de los objetos pueda tanto leer el valor como modificarlo. En ocasiones, sin embargo, puede interesarnos que una determinada propiedad sólo pueda ser leída, a modo informativo, pero no modificada. Aunque es menos habitual también puede darse el caso contrario, tener una propiedad que sólo pueda modificarse pero nunca leerse.

Para indicar que una propiedad será sólo de lectura hay que disponer el modificador `ReadOnly` delante de la palabra `Property`. En el cuerpo de la propiedad existirá tan sólo el apartado `Get`, facilitando la recuperación del valor. De manera análoga, existe el modificador `WriteOnly` para aquellas propiedades que sólo sean de escritura. En el código siguiente puede ver una clase con dos propiedades, siendo una de ellas de sólo lectura.

```
Module Module1

  Sub Main()
    Dim MiFicha As Ficha = New _
      Ficha("Francisco Charte", "El Almendral")
    Console.WriteLine("{0} -> {1}", _
       MiFicha.Nombre, MiFicha.Direccion)
    MiFicha.Direccion = "El Nogal"
    Console.WriteLine("{0} -> {1}", _
      MiFicha.Nombre, MiFicha.Direccion)

  End Sub
End Module

Class Ficha
  Private Pnombre, Pdireccion As String

  Private Sub New()

  End Sub
```

```
    Public Sub New(ByVal N As String, _
                   ByVal D As String)
      Pnombre = N
      Pdireccion = D
    End Sub

    Public ReadOnly Property Nombre() As String
      Get
        Return Pnombre
      End Get
    End Property

    Public Property Direccion() As String
      Get
        Return Pdireccion
      End Get
      Set(ByVal Value As String)
        Pdireccion = Value
      End Set
    End Property
  End Class
```

Observe que el constructor por defecto, aquel que no toma parámetros y crea un objeto vacío o con sus miembros de datos sin un valor inicial, se ha declarado como privado. Con esto evitamos que pueda crearse un objeto de la clase `Ficha` en el que el `Pnombre` quede sin un valor inicial, puesto que la propiedad `Nombre` no permite una asignación posterior. Dicho de otra forma, estamos obligando a usar el segundo de los constructores. Al ejecutar este código debería obtener un resultado similar al de la figura 8.1.

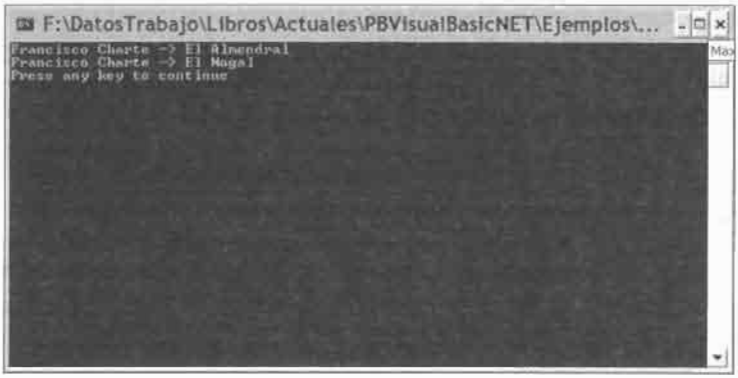


Figura 8.1. Valores almacenados en las propiedades del objeto

Lógicamente, en los métodos de lectura y asignación puede efectuarse cualquier proceso que se considere necesario, no limitándonos a devolver o guardar el nuevo valor como hemos hecho en este ejemplo.

Propiedades con índices

En caso necesario, una propiedad puede tomar parámetros que, habitualmente, se utilizan para indexar el valor que quiere recuperarse o modificarse. Pueden ser varios y de cualquier tipo, aunque lo más corriente es que sea uno de tipo numérico. El parámetro, y su tipo, será especificado entre paréntesis detrás del nombre de la propiedad. En los apartados `Get` y `Set` puede ser utilizado como convenga.

El tipo de los parámetros, como se ha dicho, puede ser cualquiera, incluso uno creado por nosotros. Podríamos, por ejemplo, definir una enumeración y utilizar sus valores como índice de una propiedad. El código siguiente corresponde a una clase con dos propiedades: `Anotacion` y `Elementos`. La primera acepta como parámetro un entero indicando el número de anotación a modificar o leer, mientras que la segunda, sólo de lectura, facilita el número máximo de anotaciones que podemos tener.

```
Class Anotaciones

  Private Panotacion() As String


  Public Sub New(ByVal Elementos As Integer)

    ReDim Panotacion(Elementos - 1)
  End Sub

  Public Property Anotacion(ByVal Indice As Integer) As String
    Get ' devolvemos la anotacion
      Return Panotacion(Indice)
    End Get
    Set(ByVal Value As String)
      Panotacion(Indice) = Value
    End Set
  End Property



  Public ReadOnly Property Elementos() As Integer
    Get
      Return Panotacion.GetLength(0)
    End Get
  End Property
End Class
```

Observe que los objetos de esta clase pueden ser usados para almacenar cualquier número de anotaciones, ya que dicho número se establece dinámicamente en el momento de creación del objeto. En el siguiente fragmento, por ejemplo, se permite al usuario indicar el número de anotaciones que va a efectuar, tras lo cual se le piden los datos y, finalmente, se enumeran.

```
Dim MisAnotaciones As Anotaciones
Dim Indice As Integer
Dim Anotacion As String

Console.Write("¿Cuántas anotaciones quieres efectuar? :")
Indice = Convert.ToInt32(Console.ReadLine())

MisAnotaciones = New Anotaciones(Indice)

Console.WriteLine("Introduce anotaciones y pulsa <Intro>")

For Indice = 0 To MisAnotaciones.Elementos - 1
  Anotacion = Console.ReadLine
  MisAnotaciones.Anotacion(Indice) = Anotacion
Next

Console.WriteLine("----------------------------------------")
Console.WriteLine("ANOTACIONES EFECTUADAS")

For Indice = 0 To MisAnotaciones.Elementos - 1
  Console.WriteLine(MisAnotaciones.Anotacion(Indice))
Next
```

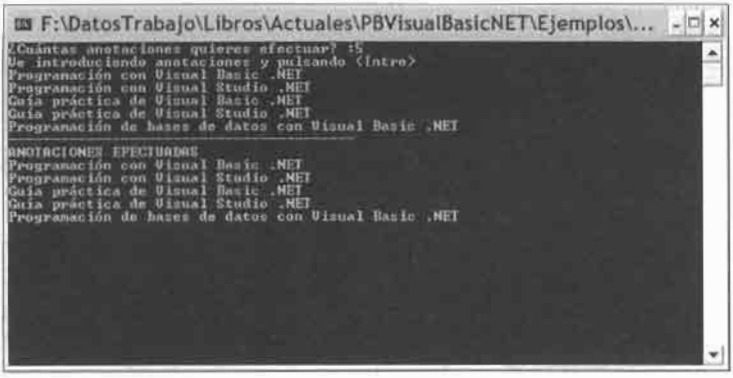


Figura 8.2. Resultado de la ejecución del código de ejemplo

La propiedad por defecto

Toda clase de componente puede disponer de una propiedad por defecto, una propiedad cuyo nombre no tiene que introducirse explícitamente detrás del punto y el nombre de los objetos para poder acceder a ella. Esta posibilidad es especialmente interesante cuando se convierte en propiedad por defecto una que acepta parámetros, ya que simplifica algo la notación final.

Para indicar que una determinada propiedad será la propiedad por defecto basta con usar el modificador `Default`. Como puede suponer, tan sólo podemos utilizarlo una vez por clase dado que no pueden existir varias propieda-

des por defecto. Tomando como base la clase usada como ejemplo en el punto anterior, podríamos modificar la definición de la propiedad `Anotacion` para dejarla así:

```
Default Public Property Anotacion(ByVal Indice As Integer) _
        As String
  Get
    Return Panotacion(Indice)
  End Get
  Set(ByVal Value As String)
    Panotacion(Indice) = Value
  End Set
End Property
```

Como se aprecia, la única diferencia está en la cabecera de la propiedad: la palabra `Default` aparece al principio. Lo demás permanece igual. Ahora, al acceder a la propiedad `Anotacion` de un objeto `Anotaciones` podríamos escribir lo siguiente:

```
Console.WriteLine(MisAnotaciones(Indice))
```

En lugar de:

```
Console.WriteLine(MisAnotaciones.Anotacion(Indice))
```

No cabe duda de que la notación es más clara y breve.

Delegados y eventos

Que un programa cree un objeto de una cierta clase y efectúe llamadas a sus métodos es algo muy usual, tanto que no pensamos en el mecanismo utilizado para enlazar la aplicación con el método de la clase. Cuando lo que se necesita es justamente lo contrario, que sea el objeto quien llame a un método del programa que lo ha creado, entonces sí que nos paramos a pensar. Este tipo de llamadas, inversas a las que hemos efectuado hasta ahora, se conoce habitualmente como *callback* o de retorno.

Los entornos de programación visuales o RAD, como Visual Basic, introdujeron hace tiempo el concepto de *evento*, entendiendo como tal una señal generada por un objeto o componente. Esta señal puede conectarse a un *gestor de evento*, un método que se ejecutaría automáticamente al recibir esa señal. Cada evento tiene una cierta firma o *tipo* que determina los parámetros que le acompañan.

En Visual Basic 6.0 y anteriores, no obstante, todo este mecanismo queda oculto para el programador.

En Visual Basic .NET, para definir el tipo de un evento se define un *delegado*, que no es más que una declaración en la que se asigna un nombre a ese tipo y se facilita la lista de parámetros. Suponiendo que fuésemos a crear un componente

que generase periódicamente un evento, como si fuese un tic de reloj, podríamos definir el delegado siguiente:

```
Public Delegate Sub OnTickReloj( _
                    ByVal Ticks As Integer)
```

Fíjese en que, de no ser por la palabra clave `Delegate`, esta declaración es idéntica a la de la cabecera de inicio de cualquier método.

Cualquier clase que desee generar un evento de este tipo definiría un campo usando la palabra clave `Event` seguida del tipo de delegado y el nombre que tendrá el evento en dicha clase. En el fragmento siguiente, por ejemplo, la clase `Reloj` contaría con un evento llamado `Tick` del tipo `OnTickReloj`. Esto significa que dicho evento irá acompañado de un parámetro entero que se facilitará al gestor que desee recibirlo.

```
Public Class Reloj
   Event Tick As OnTickReloj
```

Esta misma clase, `Reloj`, cuando quisiera generar el evento se limitaría a utilizar la instrucción `RaiseEvent` seguida del identificador, `Tick` en este caso, y, entre paréntesis, la lista de parámetros que tenga que facilitarse.

La parte complementaria a la clase que genera el evento es la clase que lo recibe, disponiendo para ello un gestor de evento apropiado con la lista de parámetros que corresponda. Para asignar el evento se usa la sentencia `AddHandler`, como puede verse en el siguiente programa completo. En él la clase `Reloj` genera el evento `Tick` descrito, que es recibido por la clase `Aplicacion`. En principio se llama al método `Inicio()` sin haber asignado un gestor de eventos, por lo que el evento `Tick` en realidad no llega a producirse. A continuación se efectúa dicha asignación y se llama de nuevo a `Inicio()`.

```
Public Delegate Sub OnTickReloj( _
                    ByVal Ticks As Integer)

Public Class Reloj
  Event Tick As OnTickReloj


  Public Sub New()

  End Sub


  Public Sub Inicio()
    Dim Ahora, Segundo As DateTime


    Ahora = Now
    Segundo = Now
```

```
      Do While Now.Subtract(Ahora).Minutes < 1

        If Now.Subtract(Segundo).Seconds = 1 Then

          RaiseEvent Tick(Now.Subtract(Ahora).Seconds)
          Segundo = Now
        End If
      Loop
    End Sub
  End Class


  Class Aplicacion

    Sub New() ' creamos un reloj
      Dim MiReloj As Reloj = New Reloj()


      Console.WriteLine( _
         "Antes de asignar el gestor de evento")
      MiReloj.Inicio()


      AddHandler MiReloj.Tick, _
          New OnTickReloj(AddressOf TickReloj)
      Console.WriteLine( _
         "Despues de asignar el gestor de evento")
      MiReloj.Inicio()
    End Sub


    Sub TickReloj(ByVal Ticks As Integer)

      Console.WriteLine(Ticks)
    End Sub

    Shared Sub Main()
      Dim MiAplicacion As Aplicacion = _
                    New Aplicacion()
    End Sub
  End Class
```

Observe cómo se usa el operador `AddressOf` para obtener la dirección de un objeto de una clase, en este caso el método `TickReloj` de la clase `Aplicacion`, y facilitarla como parámetro para crear el nuevo gestor de evento.

Definición implícita del delegado

Los delegados pueden utilizarse en diversos contextos ya que pueden alojar la dirección de cualquier método de cualquier clase, independientemente de que esté asociado a un objeto o sea un método compartido. Cuando la definición del delegado se efectúa exclusivamente para servir como tipo base de un evento, tal como en el ejemplo del punto anterior, podemos ahorrar algo de trabajo

dejando que sea la propia sentencia `Event` la que efectúe implícitamente la definición de dicho delegado.

Figura 8.3. Salida generada por el programa `Eventos`

Tomando como punto de partida el ejemplo anterior, podríamos eliminar la definición del delegado y modificar la del evento dejándola así:

```
Event Tick(ByVal Ticks As Integer)
```

En lugar de indicar el tipo del evento, como hacíamos antes, ahora especificamos la lista de parámetros que llevará asociada. Dado que no existe el tipo de delegado, la sentencia en que se añadía el controlador de evento quedaría ahora de la siguiente forma:

```
AddHandler MiReloj.Tick, AddressOf TickReloj
```

Al igual que ocurre en la declaración del evento, en esta sentencia se crea implícitamente un objeto derivado de `System.Delegate`, del que derivan todos los tipos de delegado, y se asocia con la dirección del método `TickReloj`.

El resultado final, tras efectuar los cambios indicados, es exactamente el mismo, pero el código resulta algo más breve y posiblemente más claro.

Asociación entre eventos y controladores

La asociación entre un determinado evento y su controlador o gestor, establecida mediante `AddHandler`, no tiene por qué ser perenne. En cualquier momento podemos disociar esos elementos, evento y controlador, utilizando la instrucción `RemoveHandler`.

Los parámetros necesarios son exactamente los mismos: el evento y la dirección del método.

```
RemoveHandler MiReloj.Tick, AddressOf TickReloj
```

Esta sentencia, por ejemplo, desactivaría la llamada al método `TickReloj()` cada vez que se produjese el evento `Tick` del objeto `MiReloj`.

Existe una alternativa al uso de `AddHandler` para establecer la conexión entre un evento de un objeto y un método de otro, consistente en declarar la variable de ese objeto con la cláusula `WithEvents`, palabra clave ya existente en versiones previas de Visual Basic. Declarada la variable usando ese modificador, es posible utilizar la palabra `Handles` al final de cualquier método para asociarlo con el evento.

Nota

La cláusula `WithEvents` no puede usarse en variables locales a un método, por lo que para usarla debe cambiar el ámbito de la variable `MiReloj` sacándola del constructor y dejándola al nivel de la clase `Aplicacion`.

Volviendo de nuevo sobre el mismo ejemplo anterior, podríamos modificar la clase `Aplicacion` dejándola como puede verse a continuación. Ahora el evento se gestionaría en el método `TickReloj` desde un primer momento, sin necesidad de utilizar `AddHandler`.

```
Class Aplicacion

  Dim WithEvents MiReloj As Reloj = New Reloj()

  Sub New()
    MiReloj.Inicio()
  End Sub


  Sub TickReloj(ByVal Ticks As Integer) Handles MiReloj.Tick

    Console.WriteLine(Ticks)
  End Sub
```

Como verá en capítulos posteriores, al usar diseñadores como el de formularios Windows se generará automáticamente el código necesario para enlazar los eventos y sus correspondientes gestores usando `WithEvents` y `Handles`, en lugar de `AddHandler`.

Otros usos de los delegados

Como se apuntaba anteriormente, los delegados no sólo son útiles para definir eventos, siendo también aplicables en escenarios en los cuales se precisa

la dirección de una función con el fin de invocarla sin conocer su identificador de antemano. En este contexto, y en lenguajes como C++, se usan los punteros a funciones. El inconveniente es que los punteros no suelen conservar ni comprobar información de tipos, de tal manera que un puntero puede utilizarse para llamar de manera incorrecta a una función.

Los delegados de Visual Basic .NET son unos objetos derivados de la clase `System.Delegate` y cuentan con una fuerte comprobación de tipos, es decir, no son meros punteros a funciones. Estos delegados, además, pueden tener conexión con múltiples destinos en lugar de con uno sólo. Usando un delegado es posible, por ejemplo, codificar implementaciones genéricas de procesos que, de otra manera, requerirían mucho más trabajo.

Suponga que tiene que codificar un proceso consistente en recuperar una serie de datos, tratarlos y mostrarlos por la consola. En unas ocasiones esos datos vendrán desde la consola, en otros desde un archivo, en otros de un arreglo constante de valores, en otros de un servidor de Internet, etc. En lugar de implementar un proceso específico para cada caso, que podría ser nuestra primera idea, podríamos optar por una implementación genérica:

```
Public Shared Sub Procesa(ByVal FSolicitud As SolicitaDato)
  Dim Datos(4) As String
  Dim Indice As Integer


  For Indice = 0 To 4
    Datos(Indice) = FSolicitud()
  Next


  For Indice = 0 To 4
    Console.WriteLine(Datos(Indice))
  Next
End Sub
```

Como puede ver, para solicitar los datos se invoca como función al parámetro `FSolicitud`, de tipo `SolicitaDato`. Se trata de un delegado definido previamente así:

```
Public Delegate Function SolicitaDato() As String
```

Para poder invocar a `Procesa()`, por tanto, es necesario facilitar como parámetro la dirección de una función que retorne una cadena. De dónde la obtenga no es importante. El método `Procesa()` teóricamente tendría que tratar los datos, si bien en este caso nos limitamos a mostrarlos por la consola tal y como los recuperamos.

Completemos el código anterior definiendo una clase que, además de `Procesa()`, cuente con otros dos métodos compartidos: `SolicitaTeclado()` y `SolicitaConstante()`. El primero de ellos devuelve un dato solicitado por la consola, mientras que el segundo lo obtiene de un arreglo constante de cadenas.

```
Public Delegate Function SolicitaDato() As String

Public Class ProcesaDatos

  Public Shared Sub Procesa(ByVal FSolicitud As SolicitaDato)
    Dim Datos(4) As String
    Dim Indice As Integer

    For Indice = 0 To 4
      Datos(Indice) = FSolicitud()
    Next

    For Indice = 0 To 4
      Console.WriteLine(Datos(Indice))
    Next
  End Sub

  Public Shared Function SolicitaTeclado() As String
    Console.Write("Introduzca un dato: ")
    Return Console.ReadLine()
  End Function

  Public Shared Function SolicitaConstante() As String
    Static Contador As Integer = -1
    Dim Valores() As String = _
        {"Enero", "Febrero", "Marzo", "Abril", "Mayo"}

    Contador = IIf(Contador = Valores.GetUpperBound(0), _
         0, Contador + 1)

    Return Valores(Contador)
  End Function

End Class
```

Al ser compartidos los tres métodos de esta clase, no es necesario crear un objeto para poder invocarlos. Como se ve en el código siguiente, basta con anteponer el nombre de la clase para ejecutarlos. El resultado de la ejecución es el mostrado en la figura 8.4.

```
ProcesaDatos.Procesa(AddressOf ProcesaDatos.SolicitaTeclado)
ProcesaDatos.Procesa(AddressOf ProcesaDatos.SolicitaConstante)
```

Componentes .NET

Las propiedades y los eventos, conjuntamente con los elementos que se describieron en el capítulo previo, tienen, básicamente, un objetivo: facilitar la

creación de componentes reutilizables. Los ejemplos propuestos hasta ahora no muestran este objetivo ya que su finalidad es meramente demostrativa. Lo primero que hay que hacer es identificar las funciones que van a necesitarse de manera repetitiva durante el desarrollo de un proyecto para, a continuación, implementarlas en forma de componentes. Éstos pueden ser genéricos o bien componentes específicos para aplicaciones Windows o Web.

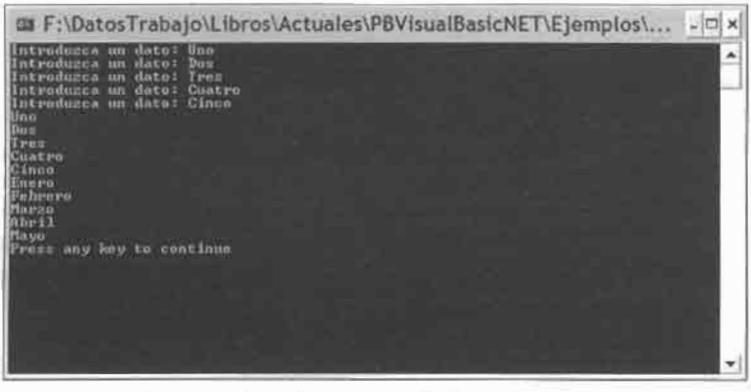


Figura 8.4. Parte de los datos se solicitan por consola

En capítulos posteriores nos ocuparemos con profundidad del desarrollo de componentes de todo tipo. No obstante, veamos brevemente cómo convertir en componente genérico una de las clases usadas previamente como ejemplo, con leves modificaciones, y cómo usarlo desde una aplicación Windows y otra de consola.

Clases y componentes

Los componentes son objetos creados a partir de una clase, como cualesquiera otros. Existen, sin embargo, algunas diferencias que permiten que esos objetos puedan ser manipulados de manera visual en un entorno como el de Visual Studio .NET, diferencias que encontramos en la interfaz `IComponent`. Ésta debe ser implementada por cualquier clase que pretenda servir como base para la creación de componentes.

En lugar de implementar manualmente `IComponent`, podemos crear una nueva clase de componente tomando como base la clase `Component`. Ésta, definida en `System.ComponentModel`, facilita una implementación por defecto de `IComponent`. Simplemente con esto ya estamos en disposición de crear una clase de componente: `Reloj`. Su finalidad la explicada anteriormente en un ejemplo: producir un evento periódicamente durante un cierto intervalo de tiempo. La única diferencia, como se aprecia en el código siguiente, es que tanto la duración del proceso como el intervalo del evento son parámetros configurables mediante sendas propiedades. El código debe introducirlo tras iniciar una nueva biblioteca de clases .NET, tal y como se hace en la figura 8.5.

Figura 8.5. Los componentes .NET se alojan en bibliotecas de clases

```
Public Class Reloj
  Inherits System.ComponentModel.Component

  Private Ptiempo, Pintervalo As Integer

  Public Event Tick(ByVal Segundos As Integer)

  Public Sub New()
    Ptiempo = 1
  End Sub

  Public Sub New(ByVal Tiempo As Integer)
    Ptiempo = Tiempo
  End Sub

  Public Property Tiempo() As Integer
    Get
      Return Ptiempo
    End Get
    Set(ByVal Value As Integer)
      Ptiempo = Value
    End Set
  End Property

  Public Property Intervalo() As Integer
    Get
      Return Pintervalo
    End Get
    Set(ByVal Value As Integer)
      Pintervalo = Value
```

```
      End Set
    End Property

    Public Sub Inicio()
      Dim Ahora, Segundo As DateTime

      Ahora = Now
      Segundo = Now

      Do While Now.Subtract(Ahora).Minutes < Tiempo

        If Now.Subtract(Segundo).Seconds = Intervalo Then

          RaiseEvent Tick(Now.Subtract(Ahora).Seconds)
          Segundo = Now
        End If
      Loop
    End Sub
  End Class
```

Observe que la clase cuenta con dos constructores. El segundo es opcional, pero el constructor por defecto es indispensable si pretendemos que el componente pueda ser manipulado visualmente en un diseñador. A pesar de que `Reloj` deriva ahora de `ComponentModel.Component`, nada nos impide seguir utilizando esta clase para crear componentes como lo hemos hecho hasta ahora, en los ejemplos previos del capítulo.

Uso no visual

La clase `Reloj`, como acaba de decirse, puede seguir utilizándose de forma no visual, por ejemplo desde una aplicación de consola. Es algo que ya sabemos cómo hacer.

El acceso a las propiedades es simple, y los eventos se conectan mediante `AddHandler` y `AddressOf` o bien mediante `WithEvents` y `Handles`. La única diferencia es que ahora no vamos a usar una clase definida en el mismo módulo, sino un componente existente en una biblioteca externa.

Tras iniciar nuestra aplicación de consola, lo primero que tenemos que hacer es pulsar el botón secundario del ratón sobre la carpeta de referencias que aparece en el Explorador de soluciones, seleccionando la opción Agregar referencia del menú emergente. En la ventana que se abre tenemos dos alternativas: usar la página **.NET** localizando el ensamblado que contiene el componente o, si tenemos abierto el proyecto de la biblioteca recién creada, recurrir a la página **Proyectos**. Esto es lo que se ha hecho en la figura 8.6, en la que se ha seleccionado el único elemento disponible. De esta forma tendremos acceso desde el nuevo proyecto a los elementos existentes en `Componentes` que, en este caso concreto, es sólo la clase `Reloj`.

Contando con esta referencia, el código necesario para crear el componente y utilizarlo no difiere mucho del de ejemplos previos. A continuación puede

ver el código completo de una aplicación consola simple que solicita la duración y el intervalo y pone en marcha el proceso.

```
Module Module1

  Dim WithEvents MiReloj As Componentes.Reloj

  Sub Main()
    Dim Tiempo As String


    Console.Write("Duración del proceso (minutos) :")
    Tiempo = Console.ReadLine()

    MiReloj = New Componentes.Reloj(Tiempo)

    Console.Write("Intervalo para el evento (segundos) :")
    MiReloj.Intervalo = Console.ReadLine()
    MiReloj.Inicio()
  End Sub


  Private Sub Reloj_Tick(ByVal Segundos As Integer) _
      Handles MiReloj.Tick
    Console.WriteLine(Segundos)
  End Sub

End Module
```

El resultado de la ejecución de este ejemplo es fácilmente imaginable y no difiere demasiado del mostrado en la figura 8.3.

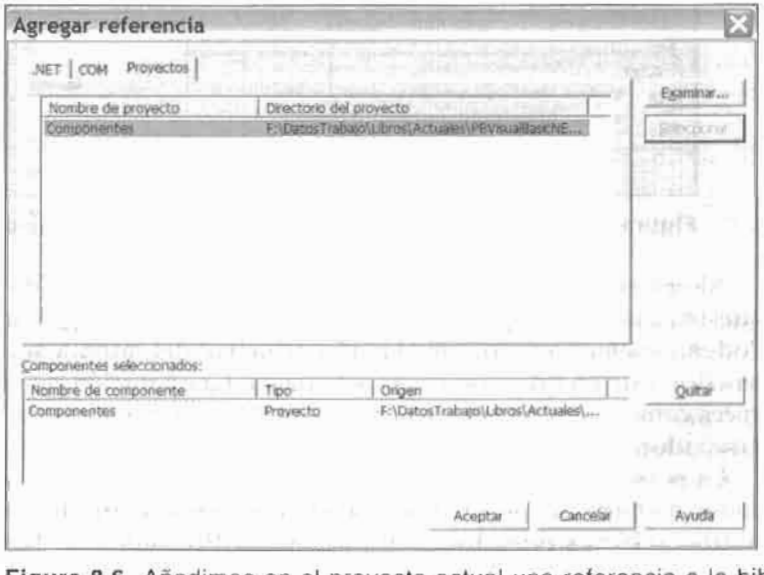


Figura 8.6. Añadimos en el proyecto actual una referencia a la biblioteca de componentes

Uso visual del componente

Derivar la clase `Reloj` de `ComponentModel.Component` e introducirla en una biblioteca de clases no nos ha aportado, aparentemente, ningún beneficio considerable. Lo que ocurre es que al crear una aplicación de consola no disponemos de un contenedor de componentes ni un diseñador para personalizarlos, elementos que sí existen, por ejemplo, en las aplicaciones Windows. Por ello vamos a iniciar un proyecto de este tipo y dar los pasos que se describen en los párrafos siguientes.

Tras iniciar el nuevo proyecto nos encontramos con un formulario vacío, el contenedor, rodeado de elementos como el Cuadro de herramientas o la ventana Propiedades, que forman parte del diseñador. Pulsamos el botón secundario del ratón sobre el Cuadro de herramientas y luego seleccionamos la opción Personalizar Cuadro de herramientas. En la ventana que aparece, concretamente en la página **Componentes de .NET Framework** (véase figura 8.7), pulsamos el botón **Examinar** y seleccionamos el ensamblado que contiene el componente `Reloj`. Dicho ensamblado aparecerá en la lista y podremos seleccionarlo.

Figura 8.7. Añadimos el componente al Cuadro de herramientas

Ahora el componente `Reloj` aparecerá en el Cuadro de herramientas, en nuestro caso, como puede verse en la figura 8.8, en la página **Componentes**. Podemos seleccionarlo, con el botón principal del ratón, y soltarlo sobre el contenedor, que en este caso es el formulario. La operación puede repetirse tantas veces como necesitemos, creando múltiples componentes de la misma clase. El diseñador asignará automaticamente un nombre para cada uno de ellos.

La personalización del componente, estableciendo valores para las propiedades `Intervalo` y `Tiempo`, puede efectuarse como hasta ahora, mediante código, o bien sirviéndonos de la ventana Propiedades. Teniendo elegido un componente `Reloj` dicha ventana mostrará todas sus propiedades. En la figura 8.9 hemos seleccionado que durante un tiempo de 1 minuto se genere el

evento cada 2 segundos. De esta forma efectuamos de manera visual un trabajo que, hasta ahora, requería la codificación de al menos dos sentencias.

Figura 8.8. Nuestro componente aparece en el Cuadro de herramientas

Figura 8.9. Las propiedades del componente aparecen en la ventana Propiedades

También la asociación entre el evento `Tick` del componente y el método que actuará como controlador de dicho evento puede establecerse automáticamente. Para ello accedemos al editor de código y, de las listas que hay en la parte superior, seleccionamos `Reloj1` en la de la izquierda y el evento `Tick` de la de la derecha. Esto provocará la inserción del correspondiente método controlador del evento, de tal manera que tan sólo tenemos que añadir el código que corresponda.

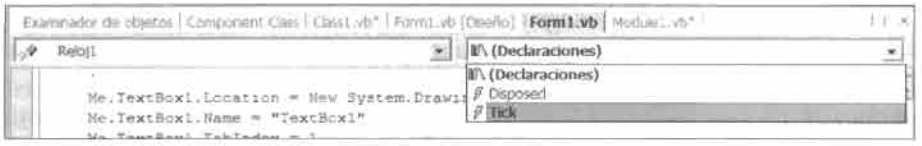


Figura 8.10. Seleccionamos el evento que queremos controlar

Para finalizar nuestro programa tan sólo resta la inserción de algún componente más en el formulario, concretamente unas cajas de texto y un botón, y la inserción del código siguiente asociado al evento `Click` del ratón y el evento `Tick` del control `Reloj`.

```
Private Sub Button1_Click( _
  ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button1.Click
  Reloj1.Inicio()
End Sub

Private Sub Reloj1_Tick( _
  ByVal Segundos As Integer) Handles Reloj1.Tick
  TextBox1.Text = Segundos
  Me.Update()
End Sub
```

En realidad nosotros sólo escribiremos las sentencias ejecutables, dado que las cabeceras y pies de los métodos son generados automáticamente por el diseñador. Finalmente, tendremos un programa que, al ser ejecutado, mostrara una ventana similar a la de la figura 8.11. En la primera caja de texto introducimos la duración y en la segunda el intervalo del evento, durante el cual se mostrarán los segundos transcurridos en la primera de las cajas.

Figura 8.11. El programa en ejecución

Nota

A diferencia de otros componentes, como las cajas de texto o los botones, el componente `Reloj` no tiene una parte visual. Por ello al insertarlo en el formulario verá que no aparece en su interior, sino en la parte inferior del diseñador.

Puntos clave

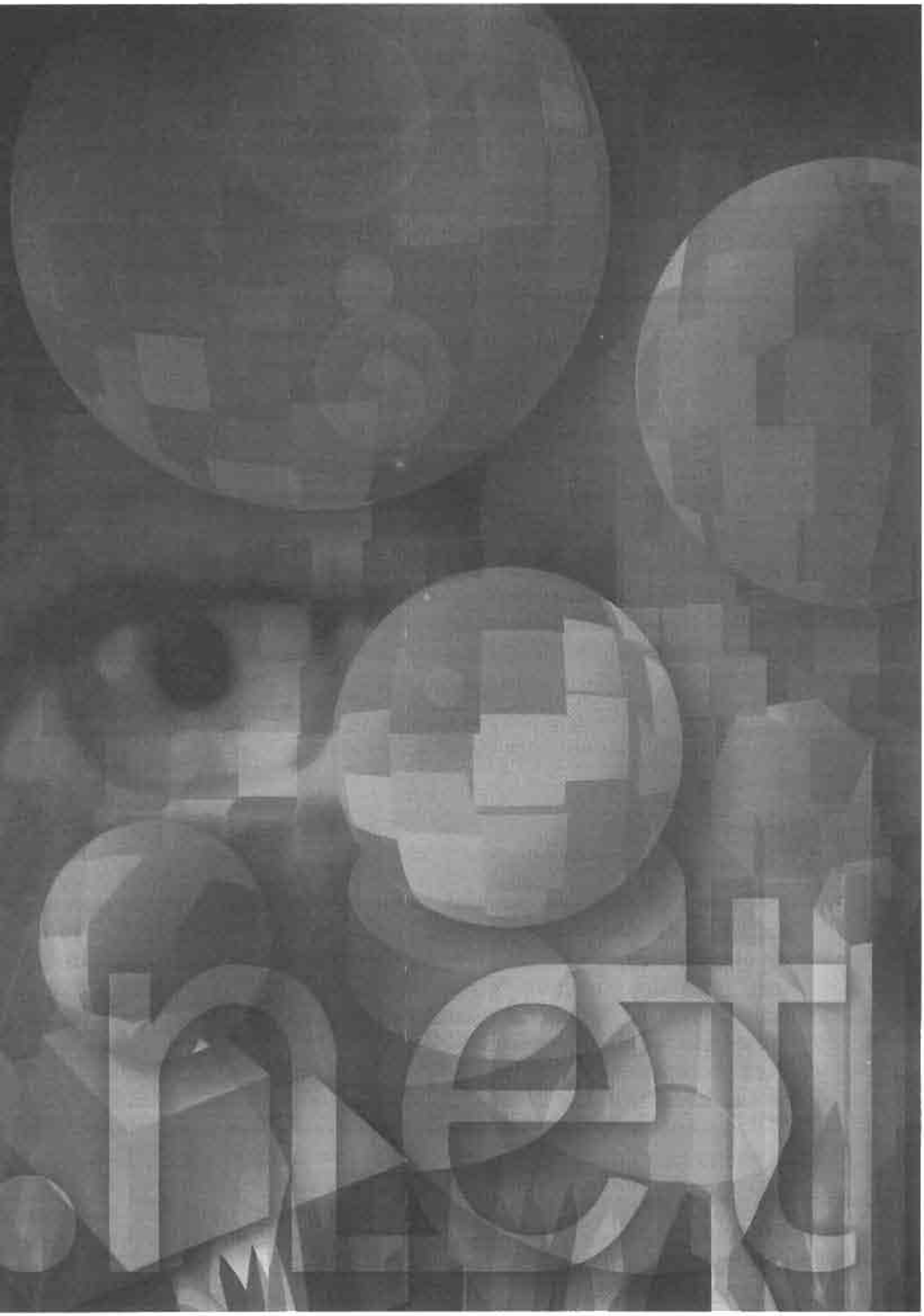
- Cualquier clase puede contar, además de con métodos y miembros de datos, con propiedades, delegados y eventos.
- Las propiedades se utilizan externamente como si fuesen variables pero, en realidad, tienen asociados métodos de asignación y lectura que controlan el proceso.
- Una propiedad puede ser sólo de lectura (`ReadOnly`), sólo de escritura (`WriteOnly`) o de lectura y escritura, según las operaciones permitidas.
- Cada clase puede contar con una propiedad por defecto (`Default`), cuyo nombre no es necesario especificar para leerla o modificarla.
- Un delegado es un tipo de dato similar a un puntero a una función pero con comprobación de tipos, puede usarse para funciones de tipo *callback* y definición de eventos.
- Mediante la palabra clave `Event` se definen los eventos con que contará una clase, generándolos mediante `RaiseEvent`.
- La conexión entre un evento, producido por un objeto, y el método que lo recibirá, en otro objeto, puede establecerse mediante `AddHandler` o bien declarando el primer objeto con `WithEvents` y añadiendo al método que lo recibe el modificador `Handles`.
- Cualquier clase derivada de `ComponentModel.Component`, que facilita una implementación por defecto de la interfaz `IComponent`, puede manipularse visualmente en un diseñador de Visual Basic .NET.

Resumen

El contenido de este capítulo, conjuntamente con el del anterior, nos ha aportado toda la información necesaria para poder definir nuestras propias clases de objetos y componentes, definiendo miembros de datos, métodos, propiedades, delegados y eventos. Visual Basic .NET dispone de un importantísimo número de clases de objetos y componentes que disponen de dichos elementos, clases que usamos creándolas desde código o bien de manera visual como se verá en capítulos posteriores.

La mayor utilidad del conocimiento adquirido, no obstante, la encontramos al definir nuestras propias clases. En ocasiones éstas se irán definiendo automáticamente, por ejemplo al añadir un formulario Windows a un proyecto e insertar componentes en su interior, mientras que en otras prácticamente todo el trabajo será manual, como al desarrollar nuevos componentes y controles.

En todo caso se trata de conceptos indispensables para poder desenvolvernos cómodamente en este Visual Basic orientado a objetos, conceptos que, además, harán mucho más fácil la comprensión de otros lenguajes, como C# o J#.


.net

9 Programación concurrente

Si pusiésemos en una lista las características que, durante años, los programadores han pedido añadir a Visual Basic 6, sin duda uno de los primeros puestos estaría ocupado por el tema que abordamos en este capítulo: la creación de componentes y aplicaciones que ejecuten de manera concurrente varias tareas, lo que se conoce habitualmente como aplicaciones con múltiples hilos de ejecución o *multithread*. Aunque existente en Visual Basic 6, esta capacidad se limita tan sólo a componentes con unas propiedades muy concretas y bastantes limitaciones.

Visual Basic .NET, al igual que el resto de los lenguajes integrados en Visual Studio .NET, puede crear libremente los hilos de ejecución que necesite. Todos los elementos necesarios se encuentran en el ámbito `System.Threading`, como podremos ver en este capítulo. La gestión de múltiples hilos, no obstante, plantea también nuevas dificultades como la sincronización a la hora de acceder a datos comunes a varios hilos. El estudio de estos problemas es uno de los objetivos que abordamos de inmediato.

Aplicaciones de la concurrencia

Que exista la posibilidad de crear y utilizar múltiples hilos de ejecución no implica que necesariamente tengamos que hacerlo, de igual manera que no utilizamos un bucle o un condicional por el simple hecho de que sean construcciones del lenguaje, sino que lo hacemos siempre a causa de una cierta necesidad.

Mientras desarrollamos una aplicación, por tanto, no tenemos que pensar dónde podemos aplicar el uso de varios hilos concurrentes, más bien lo contrario, este hecho debe surgir como una necesidad.

Hay que tener en cuenta que el rendimiento de una aplicación, en contra de la creencia general, es inversamente proporcional al número de hilos de ejecución concurrentes que utilice. Es decir, a más hilos de ejecución en paralelo peor rendimiento general. Es fácil de comprender estableciendo un simple paralelismo entre la herramienta que efectúa el trabajo, que es el procesador que hay en el ordenador, y cualquier otra herramienta con la que esté habituado a trabajar. Suponga que tiene que hacer varios agujeros en una pared utilizando una máquina taladradora. ¿Cómo tardará menos, haciendo un agujero entero detrás de otro o cambiando cada segundo de un agujero a otro continuamente hasta tenerlos terminados los tres? Esta claro que el segundo método requerirá más tiempo por el simple hecho de tener que detener la máquina y cambiar de un agujero a otro muchas más veces de las necesarias, aunque al final, aparentemente, terminemos todos los agujeros al mismo tiempo.

Cuando la tarea de un programa se divide en varias partes concurrentes ocurre prácticamente lo mismo, siempre hablando de sistemas en los que tan sólo existe un microprocesador. Este emplea parte de su tiempo en cambiar de un hilo a otro, tiempo que no puede dedicar a la ejecución de la tarea que es realmente lo importante.

Tras leer esta exposición seguramente se preguntará para qué le puede servir entonces la creación de uno o más hilos de ejecución concurrente, ya que parece no tener parte positiva. Lo explicado hasta ahora ha tenido como objetivo hacerle pensar un momento antes de decidir la utilización de varios hilos pero, como puede suponer, esta posibilidad tiene aplicaciones muy claras. Una de ellas se aplica especialmente a aquellos programas que cuentan con una interfaz de cara al usuario, típicamente una ventana, que no debe dejar de responder a causa de la puesta en marcha de un proceso más o menos lento. Ese proceso debería ejecutarse en un hilo de ejecución independiente, sin afectar a la mencionada interfaz.

Otro caso claro en el que se necesita el uso de este recurso es en las aplicaciones de servidor que deben atender simultáneamente a multitud de clientes. Lo habitual es que a medida que van recibiéndose solicitudes se lancen hilos de ejecución para atenderlas, independientes para cada cliente. Esta técnica evita que cada cliente tenga que esperar hasta que la solicitud del anterior finalice, lo cual podría dar la sensación de falta de respuesta por parte del servidor. Aunque el rendimiento global podría ser inferior, usando múltiples hilos todos los clientes tienen la impresión de que se les atiende de inmediato.

Exceptuando los dos casos indicados, los más corrientes, y alguno más concreto, el uso de la concurrencia no suele ser una buena idea en la gran mayoría de los proyectos.

Ahora que sabe esto, dediquemos el resto del capítulo a aprender a utilizar múltiples hilos de ejecución. Los ejemplos propuestos son en su mayoría muy sencillos, no representando los casos en los cuales se necesitaría realmente una ejecución concurrente de varias tareas.

Nota

En el capítulo anterior se usó como ejemplo una clase que simulaba un reloj generando un evento a intervalos. Mientras el método estaba ejecutándose el resto de la aplicación quedaba totalmente detenida. Utilizando un hilo de ejecución independientemente podríamos conseguir que el evento `Tick` se produjese de manera asíncrona, tras devolver el método `Inicia()` el control al código que le hubiese invocado.

Aplicaciones, hilos y prioridades

En el tercer capítulo se introdujeron múltiples conceptos relativos a la plataforma Microsoft .NET, entre ellos el del dominio de aplicación. Todos los proyectos desarrollados con Visual Basic .NET se ejecutan en el contexto de un dominio de aplicación, contexto representado por un objeto de la clase `AppDomain`. Podemos recuperar el dominio de aplicación actual mediante la propiedad compartida `CurrentDomain`. Con las propiedades y los métodos de `AppDomain` es posible recuperar información de contexto, abrir nuevos ensamblados, ejecutarlos, etc.

Cuando se crea un dominio de aplicación y se abre un ensamblado, para iniciar la ejecución de un programa, se crea y pone en marcha un hilo de ejecución o *thread*. Éste es el hilo de ejecución principal y, a menos que creemos otros de manera explícita, único con el que contará la aplicación. Mediante la propiedad compartida `CurrentThread` de la clase `Thread` podemos recuperar una referencia al objeto que representa a ese hilo principal.

La ejecución de un nuevo hilo concurrente consiste, básicamente, en la creación de un nuevo objeto de la clase `Thread`, su asociación con el método que contiene el código a ejecutar y la puesta en marcha mediante el método `Start()`.

La clase `Thread`

El ámbito `System.Threading` es el contenedor en el que podemos encontrar todas las clases e interfaces relacionadas con la programación concurrente en Visual Basic .NET, entre ellas la clase `Thread`. Ésta encapsula todos los recursos necesarios para gestionar un hilo de ejecución: creación, puesta en marcha, pausa, reanudación y detención. La clase `Thread` es de uso final, es decir, no puede ser utilizada como base para crear otras nuevas.

Mediante las propiedades de la clase `Thread` podemos saber si el hilo está o no ejecutándose, si lo hace en segundo plano o cuál es su prioridad, mientras que mediante los métodos se habilitan funciones como la puesta en marcha, suspensión temporal, interrupción o enlace con otro hilo de ejecución. Los mecanismos de sincronización, algunos de los cuales conoceremos después, no forman parte de la clase `Thread` sino que aparecen como objetos independientes de otras clases del ámbito `System.Threading`.

Para crear un objeto de la clase `Thread` es necesario, así lo requiere el único constructor disponible, entregar un parámetro de tipo `ThreadStart`. Dicho tipo se encuentra definido en el mismo ámbito `System.Threading` de la siguiente manera:

```
Public Delegate Sub ThreadStart()
```

Como puede ver se trata de un delegado, concretamente una referencia a un procedimiento que no toma parámetros. Al crear un objeto `Thread`, por tanto, deberemos entregar como parámetro la dirección de un método que se ajuste a la definición de ese delegado:

```
Dim MiHilo As Thread = New Thread(AddressOf MiMetodo)
...
Public Sub MiMetodo()
    ...
End Sub
```

En este caso el objeto `MiHilo`, que representa un hilo de ejecución, está asociado al método `MiMetodo()`. Cuando el hilo se ponga en marcha ejecutará el código que exista en ese método.

Puesta en marcha del nuevo hilo

La creación de un nuevo hilo de ejecución no implica la ejecución inmediata del código asociado, contenido, como acaba de decirse, en el método cuya dirección se entrega como parámetro al constructor de la clase `Thread`. La puesta en marcha del nuevo hilo es un proceso independiente, de tal manera que podemos crear tantos hilos como necesitemos dejándolos en espera hasta el momento en que nos interese ejecutarlos.

En la definición de la clase `Thread` encontramos métodos como `Start()`, `Suspend()`, `Resume()` y `Abort()`. Su propio nombre denota la funcionalidad: poner en marcha el hilo de ejecución, suspenderlo temporalmente, reanudarlo o bien detenerlo de manera definitiva. Mediante la propiedad sólo de lectura `ThreadState` podemos conocer el estado actual del hilo. Dicha propiedad contendrá uno de los valores enumerados en la tabla 9.1. También podemos servirnos de propiedades como `IsAlive` o `IsBackground`, que devuelven `True` en caso de que el hilo esté en ejecución o desempeñe su tarea en segundo plano, respectivamente.

Tabla 9.1. Valores de la enumeración `ThreadState`

Valor	Estado del hilo
`Unstarted`	Aún no se ha puesto en marcha
`Running`	Está ejecutándose
`SuspendRequested`	Se ha solicitado su detención temporal

Valor	Estado del hilo
`Suspended`	Se encuentra detenido temporalmente
`StopRequested`	Se ha solicitado su detención
`Stopped`	Se encuentra detenido
`AbortRequested`	Se ha solicitado su interrupción
`Aborted`	Interrumpido
`WaitSleepJoin`	Se encuentra bloqueado o en espera
`Background`	Está ejecutándose como hilo en segundo plano

Tras crear el objeto `Thread`, por tanto, lo único que necesitamos para poner en marcha la ejecución del hilo es llamar al método `Start()`. Éste devolverá el control de manera inmediata, de tal forma que el hilo de ejecución principal pueda continuar con su trabajo. Mientras tanto, el nuevo hilo hará su trabajo en paralelo, al menos aparentemente.

Enumeraciones concurrentes

Para ver en la práctica lo aprendido hasta ahora vamos a servirnos de un ejemplo simple pero demostrativo. En él pondremos en marcha dos hilos de ejecución adicionales al principal, cada uno de ellos efectuando una enumeración. Como en los capítulos anteriores, continuaremos utilizando el asistente para aplicaciones de consola con el fin de centrarnos en la funcionalidad del ejemplo olvidándonos de tareas de diseño. Lo primero que haremos es definir una clase en la que existen varios métodos encargados de enumerar valores por la consola. En el ejemplo siguiente puede ver que existen dos métodos: uno muestra una sucesión de números y otro una secuencia de caracteres:

```
Public Class Enumeraciones

  Public Sub Numeros()
    Dim Contador As Integer

    For Contador = 1 To 1000
      Console.Write("{0}{1}", Contador, vbTab)
    Next
  End Sub

  Public Sub Letras()
    Dim Letra As Integer

    For Letra = 1 To 255
      Console.Write("{0}{1}", Chr(Letra), vbTab)
    Next
  End Sub
End Class
```

Como es habitual, podemos crear un objeto de esta clase e invocar a sus métodos directamente, como hemos hecho con muchos otros objetos en ejemplos previos. El resultado es que al llamar a un método, por ejemplo `Numeros()`, será el hilo principal el que entre a ejecutarlo, de tal forma que no se devolverá el control hasta que `Numeros()` finalice. Teniendo esto en cuenta es fácil suponer el resultado de las sentencias siguientes:

```
Dim MisEnumeraciones As New Enumeraciones()

MisEnumeraciones.Numeros()
MisEnumeraciones.Letras()
Console.WriteLine("Fin del método Main()")
```

Al ponerse en marcha el programa se inicia el hilo principal de ejecución, que será el encargado de procesar estas sentencias. Cuando se llegue a `MisEnumeraciones.Numeros()` dicho hilo pasará a ejecutar las sentencias del método `Numeros()`, finalizado el cual volverá al cuerpo principal del programa y ejecutará `MisEnumeraciones.Letras()`. Resumiendo: en la consola aparecerá primero la serie de números, después la de caracteres y, finalmente, el mensaje `Fin del método Main()`.

Muy distinto será el resultado que generaría el código siguiente:

```
Sub Main()

  Dim HiloNumeros, HiloLetras As Thread

  Dim MisEnumeraciones As New Enumeraciones()

  HiloNumeros = New Thread(AddressOf MisEnumeraciones.Numeros)

  HiloLetras = New Thread(AddressOf MisEnumeraciones.Letras)

  HiloNumeros.Start()
  HiloLetras.Start()


  Console.WriteLine("Fin del método Main()")
End Sub
```

Al ejecutar la sentencia `HiloNumeros.Start()` el hilo principal pone en marcha otro hilo que será el encargado de ejecutar el código de `MisEnumeraciones.Numeros()`, ocurriendo lo mismo con `HiloLetras.Start()`. El hilo principal, por tanto, queda totalmente libre tras poner en marcha los otros dos, de tal forma que el mensaje final aparecerá, generalmente, antes de que los dos hilos independientes terminen su trabajo. En la figura 9.1 puede ver parte del resultado que genera la ejecución del programa. El primer hilo comienza a mostrar números, tras lo cual toma el control el segundo y muestra algunos caracteres. En ese momento el hilo principal llega a su fin y toma de nuevo el control el segundo de los hilos, después el primero y así irán simultaneándose hasta que ambos terminen su trabajo.

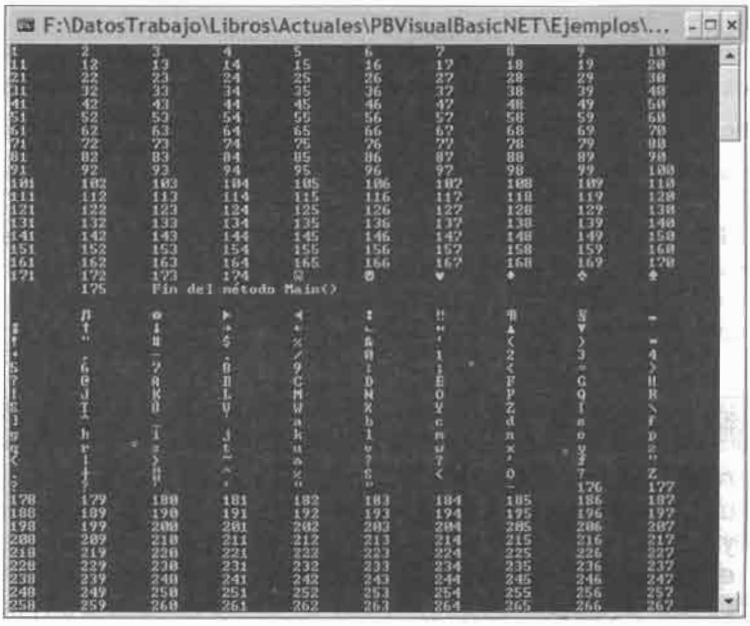


Figura 9.1. Cada hilo de ejecución va mostrando unos datos

Cada vez que ejecute el programa posiblemente vea un resultado distinto, ya que la cadencia con la que el sistema cambia de un hilo de ejecución a otro depende de múltiples factores.

Prioridad de ejecución

El tiempo que el sistema cede el control a un hilo de ejecución, así como la frecuencia con que lo hace, son factores que dependen de la prioridad de ejecución que tenga ese hilo. Cuando se crea un nuevo hilo por defecto se le asigna una prioridad *normal*, de tal forma que tiene el mismo tiempo de procesador que el hilo principal de la aplicación.

Podemos tanto establecer como determinar la prioridad de un cierto hilo mediante la propiedad `Priority`. Los valores que puede tener esta propiedad son los enumerados en la tabla 9.2. Asignando el valor `AboveNormal` al hilo `HiloNumeros`, por poner un ejemplo, conseguiremos que éste se ejecute más tiempo que los demás de la aplicación. Si le asignásemos el valor `BelowNormal`, por el contrario, el tiempo asignado sería inferior.

Tabla 9.2. Posibles valores para la propiedad `Priority`

Valor	Prioridad
`Normal`	La establecida por defecto. Prioridad media de todos los hilos en el sistema
`AboveNormal`	Por encima de la normal (más tiempo dedicado al hilo)

Valor	Prioridad
`Highest`	La más alta (el mayor tiempo posible dedicado al hilo)
`BelowNormal`	Por debajo de la normal (menos tiempo dedicado al hilo)
`Lowest`	La más baja (el menor tiempo posible dedicado al hilo)

Puede hacer algunas pruebas simples modificando el valor de la propiedad `Priority` de `HiloNumeros` e `HiloLetras` antes de llamar a sus métodos `Start()`. Dependiendo de los valores asignados verá que la salida por la consola cambia.

Nota

A pesar de que podemos asignar libremente la prioridad que deseemos a un objeto `Thread`, no es recomendable el uso de los valores `AboveNormal` y `Highest`, especialmente en varios hilos de manera simultánea dado que ello puede afectar al rendimiento global del sistema. Sí es habitual, por el contrario, asignar `BelowNormal` o incluso `Lowest` a un hilo cuya tarea no es demasiado importante en cuanto al tiempo que se empleará en llevarla a cabo.

Elementos de sincronización

La mayoría de las aplicaciones que utilizan múltiples hilos de ejecución concurrentes no son tan simples como el ejemplo anterior, de ahí que en él no encontremos, de principio, los problemas que sí se plantean en dichas aplicaciones. Uno de esos problemas, quizá el más importante, es la sincronización entre hilos de ejecución, especialmente a la hora de acceder a recursos que son compartidos entre varios de ellos o, incluso, todos.

Suponga que tiene un componente de servidor que crea un hilo de ejecución independiente para cada cliente del que se reciba una solicitud, cliente al que se asigna un número que, aunque no consecutivo, sí debe ser único. Deberemos contar, por tanto, con una variable que almacene el último índice usado, variable cuyo valor se incrementará al crear cada nuevo hilo. Tendríamos una variable compartida por todos los hilos, lo cual plantea un primer problema.

No vamos a diseñar un complejo componente de servidor para ver cuál sería su funcionamiento según el esquema que acaba de describirse, sino que utilizaremos un ejemplo bastante sencillo para simular una situación similar. El punto central de dicho ejemplo será la clase siguiente:

```
Public Class Contador
```

```
    Private N As Integer


    Public Sub Cuenta()
      N += 1


      If N Mod 5 = 0 Then Thread.Sleep(1)

      Thread.CurrentThread.Name = N
    End Sub
  End Class
```

La clase no efectúa ningún proceso complejo para elaborar una respuesta de cara al cliente, limitándose a la asignación de un índice teóricamente único. De forma pseudo-aleatoria, en realidad siempre que el resto de dividir N entre 5 sea cero, suspendemos la ejecución del hilo actual durante un milisegundo, de tal manera que, en la práctica, múltiples hilos ejecutando el método `Cuenta()` emplearán distinto tiempo en realizar su trabajo.

Observe que la variable `N` se ha declarado como privada a la clase. En el método `Cuenta()` se incrementa su valor y, a continuación, se asigna a la propiedad `Name` del hilo actual, del hilo que esté ejecutando el método en ese momento. Teóricamente el proceso es correcto, pero veamos qué ocurre al simular que se reciben solicitudes de unas decenas de clientes:

```
  Sub Main()

    Dim MiContador As New Contador()

    Dim Hilos() As Thread
    Dim Indice As Integer


    ReDim Hilos(49)

    For Indice = Hilos.GetLowerBound(0) To Hilos.GetUpperBound(0)

      Hilos(Indice) = New Thread(AddressOf MiContador.Cuenta)
      Hilos(Indice).Start()
    Next



    For Indice = Hilos.GetLowerBound(0) To Hilos.GetUpperBound(0)
      Console.WriteLine("El Hilo({0}) se llama {1}", _
          Indice, Hilos(Indice).Name)
    Next
  End Sub
```

Pude modificar la sentencia `Redim` para alterar el número de hilos que concurrentemente ejecutarán el método `Cuenta()` y comprobar las diferencias al aumentar o disminuir ese número. En la figura 9.2 puede ver un ejemplo de

dicho resultado. Observe que la mayoría de los hilos no tienen un contenido en la propiedad `Name`. Además, algunos índices no son utilizados, mientras que otros aparecen en más de un hilo. Definitivamente nuestra implementación tiene más de un problema.

Figura 9.2. Los hilos entran en conflicto y generan un resultado inesperado

Esperas entre hilos

Lo primero que nos llama la atención del resultado que hemos obtenido es el hecho de que la mayoría de los hilos no tienen un índice, es decir, la propiedad `Name` del objeto `Thread` que representa al hilo está vacía. En el primer bucle que hay en el método `Main()` se ponen en marcha varias decenas de hilos de ejecución, tras lo cual el hilo principal sigue su curso y llega al segundo de los bucles. En él se recorren nuevamente todos los hilos recuperando, y mostrando por la consola, el índice correspondiente. El problema es que cuando se llega al segundo bucle no todos los hilos de ejecución previamente iniciados han completado su ejecución, de ahí que en algunos la propiedad `Name` no tenga un valor. Antes de recuperar dicha propiedad, el hilo principal debería asegurarse de que el hilo que va a examinar haya finalizado su trabajo.

La codificación actual del segundo bucle debería modificarse, de tal forma que antes de acceder a la propiedad `Name` del hilo `Indice` estemos seguros de que ese hilo ha terminado y, por tanto, la mencionada propiedad tendrá un valor correcto. Esta modificación es realmente simple: basta con añadir la línea siguiente en el interior del bucle, justo antes de llamar a `WriteLine()`.

```
Hilos(Indice).Join()
```

Al invocar al método `Join()` de un cierto hilo, en este caso `Hilo(Indice)`, el hilo desde el que se efectúa la llamada, en nuestro caso el hilo principal de

ejecución, quedará detenido hasta que el primero llegue a su fin. En otras palabras, hacemos una pausa en la ejecución del bucle hasta que el hilo `Indice` ha terminado, momento en el cual podemos recuperar el contenido de la propiedad `Name` con la seguridad de que ya se ha efectuado la asignación.

Tras este cambio, una nueva ejecución del programa generará el resultado que puede verse en la figura 9.3. Ahora todos los hilos tienen un índice pero, como se aprecia en esa figura, existen claros conflictos. Por una parte hay varios con el mismo índice, mientras que algunos números no han llegado a utilizarse. Si utilizásemos ese índice para identificar al cliente tendríamos problemas ya que la asignación no ha sido correcta.

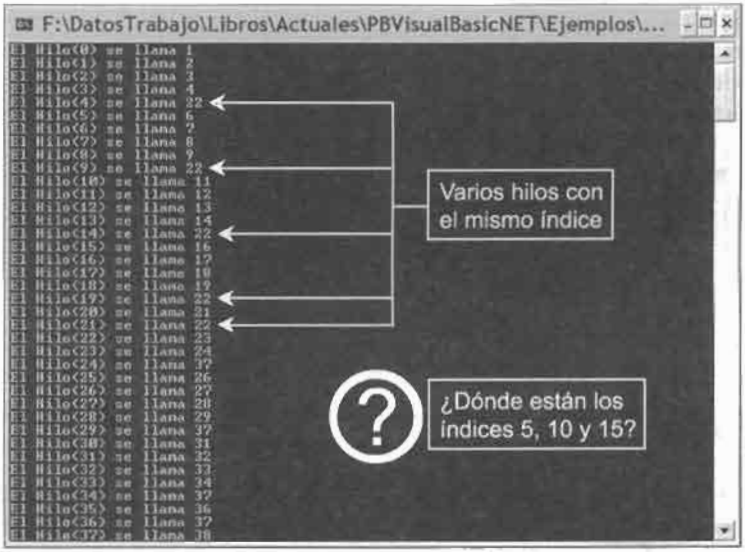


Figura 9.3. Aunque con mejoras, el resultado sigue siendo incorrecto

Accesos exclusivos

El problema que tiene nuestro programa es que múltiples hilos, de manera desordenada, modifican y recuperan el valor de la variable `N`, común a todos ellos. La causa es que el sistema puede interrumpir la ejecución de un hilo mientras está procesando las tres sentencias siguientes, dando paso a un nuevo hilo que comenzaría por la primera o continuaría donde se hubiese quedado si es que había sido suspendido previamente.

```
N += 1

If N Mod 5 = 0 Then Thread.Sleep(1)

Thread.CurrentThread.Name = N
```

Suponga que se pone en marcha el primer hilo de ejecución y, tras incrementar el valor de `N`, es suspendido para dar paso a la ejecución del segundo. Éste

vuelve a incrementar el valor N y lo toma como índice propio, tras lo cual se reactiva el primer hilo que, como puede suponer, encuentra en N el mismo valor que el segundo. Es el caso en que varios hilos toman el mismo índice. De manera análoga, debido al incremento desordenado de múltiples hilos, es posible que queden índices sin utilizar.

La solución a este problema es lógica: debemos evitar que mientras un hilo está ejecutando las tres sentencias anteriores otro hilo pueda acceder a ellas. Dicho de otra forma, el código de esas tres sentencias debería ser exclusivamente accesible para un solo hilo. Con este fin podemos utilizar diversos mecanismos, siendo uno de ellos la clase `Monitor`.

Dos de los métodos de esta clase, `Enter()` y `Exit()`, disponen una señal en un objeto que debemos facilitar como parámetro. Dicho objeto debe ser general a todos los hilos cuyo acceso quiere controlarse, pudiendo ser de cualquier tipo. Podemos utilizar, por ejemplo, la referencia al propio objeto al que pertenece el método `Cuenta()` que, tras modificarlo, quedaría así:

```
Public Sub Cuenta()
  Monitor.Enter(Me)
  N += 1
  If N Mod 5 = 0 Then Thread.Sleep(1)
  Thread.CurrentThread.Name = N
  Monitor.Exit(Me)
End Sub
```

Observe que no creamos ningún objeto de la clase `Monitor`, de hecho no es posible, sino que usamos directamente los métodos compartidos `Enter()` y `Exit()` para delimitar la porción de código que no debe ser accedida por más de un hilo simultáneamente. Tras este nuevo cambio, la ejecución del programa producirá un resultado similar al de la figura 9.4. A pesar de que la asignación de índices no es consecutiva, ya que los distintos hilos no tienen por qué ejecutarse de manera secuencial, hemos conseguido nuestro objetivo: cada objeto `Thread` tiene un índice único y, además, no hay índices sin usar. La identificación, en el hipotético componente de servidor, estaría conseguida.

Nota

En este ejemplo se ha sincronizado el acceso exclusivo de los hilos de ejecución a todo el código del método`Cuenta()`, de tal manera que se pierde, en parte, la ventaja de la ejecución simultánea que facilita el trabajo con varios hilos. En la práctica, sin embargo, las sentencias alojadas entre las llamadas a `Enter()` y `Exit()`, serían sólo las que modificasen y recuperasen el valor de N, mientras que el resto del procedimiento, que realizaría la tarea de cara al cliente, podría seguir ejecutándose de manera concurrente.

Figura 9.4. Obtenemos el resultado correcto al sincronizar adecuadamente los hilos

Otros métodos de sincronización

La clase `Monitor` es tan sólo uno de los recursos existentes en Visual Basic .NET para sincronizar el acceso a datos comunes. El propio lenguaje, así como el ámbito `System.Threading`, disponen de alternativas que permiten controlar tanto dicha situación como similares que pudieran surgir.

Uno de los recursos del lenguaje, del nuevo Visual Basic .NET, es la sentencia `SyncLock`. Ésta va emparejada con una `End SyncLock`, de manera similar a lo que ocurre con `If`, `Select` y otras sentencias que delimitan bloques de sentencias. Al igual que ocurre con la clase `Monitor`, la sentencia `SyncLock` necesita un objeto compartido entre los distintos hilos para poder señalizar la sincronización. Todas las sentencias introducidas en el bloque serán ejecutadas sólo por un hilo de manera concurrente.

Usando `SyncLock`, por tanto, podríamos modificar el fragmento de código del punto anterior dejándolo como sigue:

```
Public Sub Cuenta()
  SyncLock Me
    N += 1

    If N Mod 5 = 0 Then Thread.Sleep(1)
```

```
        Thread.CurrentThread.Name = N
    End SyncLock
```

El resultado obtenido, al ejecutar este código, es exactamente el mismo que ya teníamos. Es lógico dado que la sentencia `SyncLock` lo que hace, al fin y al cabo, es utilizar un monitor para gestionar la sincronización de las sentencias indicadas.

Otro recurso a nuestra disposición es la clase `Mutex`, cuyo funcionamiento se asemeja a la clase `Monitor` ya usada en el punto previo. En lugar de los métodos `Enter()` y `Exit()` se usan `WaitOne()` y `ReleaseMutex()`. La diferencia es que un `Mutex` permite registrar varias esperas sin necesidad de crear múltiples objetos.

Interbloqueos y señales

En una de las versiones del ejemplo que está utilizándose a lo largo del capítulo hemos usado el método `Join()` de la clase `Thread` para esperar la finalización de un cierto hilo de ejecución, de tal forma que pudiera recuperarse el resultado por el producido de una manera segura. Es ésta una técnica habitual: lanzar dos hilos de ejecución paralelos y que, en un momento determinado, uno espere a que el otro termine para poder utilizar los resultados que ha generado.

Al usar el método `Join()` corremos un riesgo si nuestra implementación no está bien diseñada: que un hilo espere la finalización de otro y que ese otro, con otra llamada a `Join()`, esté, a su vez, esperando la finalización del primero. Es lo que se conoce como un *interbloqueo* o *dead-lock*. El resultado es que los dos hilos permanecen detenidos indefinidamente causando, en la práctica, el bloqueo de la aplicación.

La implementación del método `Join()` usada en el ejemplo anterior espera hasta que el hilo termina su ejecución, pero existen otras versiones del mismo método que permiten indicar un tiempo máximo de espera, en milisegundos, transcurrido el cual la ejecución continuará a pesar de que el hilo supervisado aún no haya llegado a su fin.

En lugar de usar el método `Join()`, que podríamos considerar como el elemento de alto nivel para sincronizaciones entre hilos, podemos recurrir al uso de clases como `ManualResetEvent` y `AutoResetEvent`. Ambas gestionan la señalización necesaria para permitir que unos hilos esperen a que otros les permitan o no continuar. En la documentación del ámbito `System.Threading` encontrará la referencia completa de éstas y otras clases.

Nota

Una problemática especial se plantea al invocar a métodos de elementos de interfaz de usuario, tal como pueden ser los controles de un formulario

Windows, desde un hilo de ejecución diferente al que se utilizó para su creación. La sincronización, en este caso, pasa por el uso del método `Control.Invoke()` para llamar indirectamente al método que nos interese, en lugar de invocarlo directamente como es lo habitual.

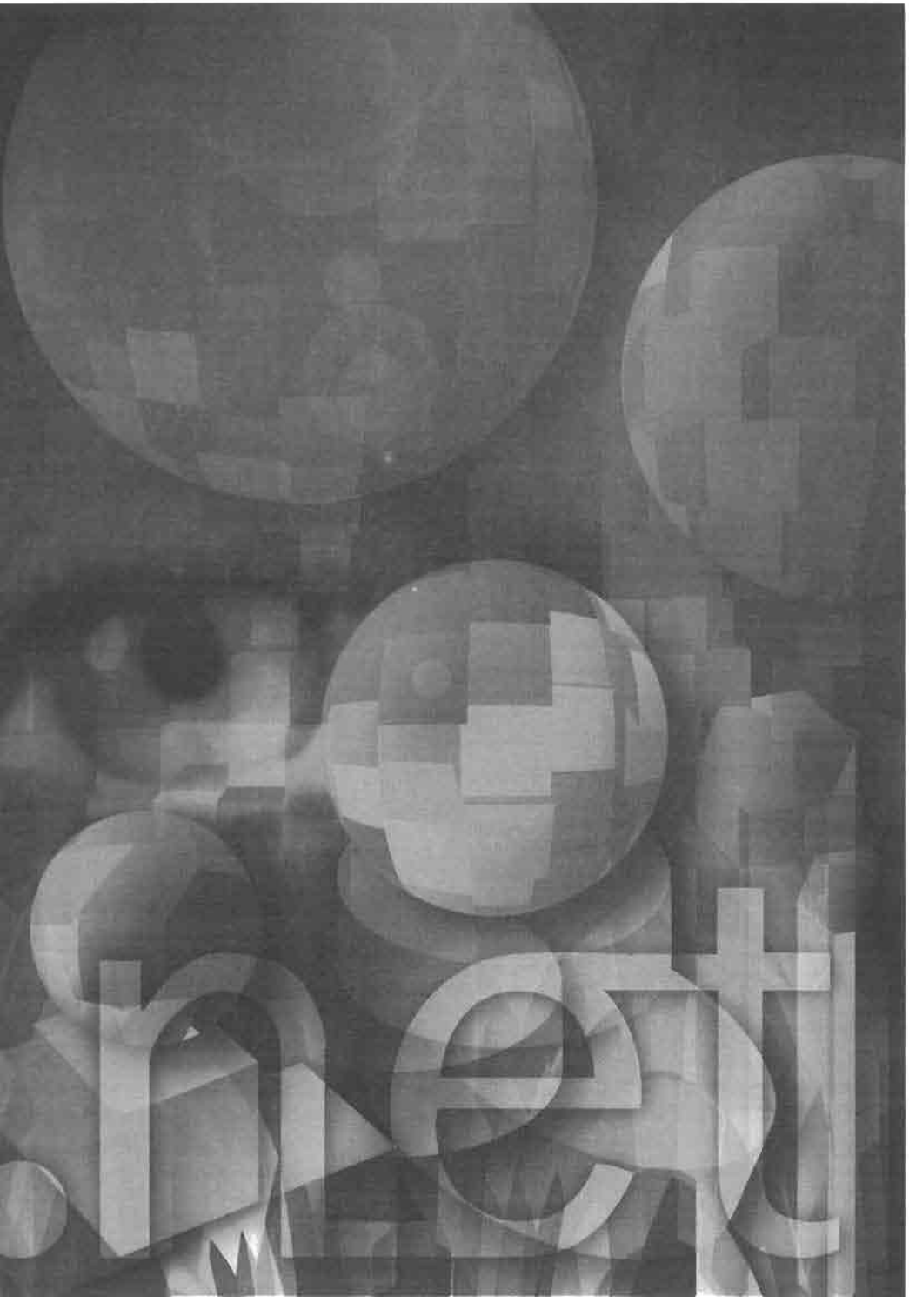
Puntos clave

- El uso de múltiples hilos de ejecución en un proyecto tiene aplicaciones muy concretas: atender simultáneamente a múltiples clientes, mantener viva una interfaz de usuario o dejar en segundo plano procesos lentos. Su uso indiscriminado afectará negativamente al rendimiento general del programa.
- Para gestionar un hilo de ejecución se usan los métodos y propiedades de la clase `Thread`.
- Al crear un objeto `Thread` es necesario facilitar la dirección de un método. Este contendrá el código a ejecutar por el hilo.
- La creación de un hilo no implica su ejecución inmediata. Disponemos de varios métodos que nos permiten controlar el hilo: `Start()`, `Suspend()`, `Resume()`, etc.
- Cada hilo tiene asociada una prioridad de ejecución que determina el tiempo de procesador que puede usar. Podemos conocer, y modificar, dicha prioridad con la propiedad `Priority` de la clase `Thread`.
- El acceso simultáneo por varios hilos de ejecución a datos compartidos puede generar problemas de sincronización, para evitarlos existen clases como `Monitor` y `Mutex`, así como nuevas sentencias como `SyncLock`.
- Un hilo puede esperar la finalización de otro de diversas maneras. Una de ellas es el método `Join()` de la clase `Thread`, otro el uso de las clases `AutoResetEvent` y `ManualResetEvent`.

Resumen

El conocimiento que ha adquirido a lo largo de este capítulo le servirá como punto de partida a la hora de desarrollar aplicaciones y componentes que usen múltiples hilos de ejecución, aunque en la práctica puede encontrarse con más problemas y precisar elementos adicionales de control o sincronización. En definitiva, ahora mismo ya sabe cómo poner en marcha varios hilos de ejecución, cómo ponerlos en marcha y determinar su estado. También ha aprendido las bases de la sincronización entre hilos a la hora de acceder a información compartida.

Tratadas en éste y los capítulos precedentes temáticas generales, como las estructuras de control, las nuevas construcciones orientadas a objetos o el mismo uso de la concurrencia, que pueden aplicarse a cualquier tipo de aplicación o componente, a partir del próximo capítulo comenzaremos a ocuparnos de temas más específicos como la construcción de aplicaciones Windows y Web basadas en formularios, la elaboración de gráficos o su impresión. En todos estos contextos necesitaremos contar con la base aportada en estos primeros capítulos, base que nos ha permitido conocer el lenguaje Visual Basic .NET y algunos servicios concretos de la plataforma .NET.



10 Formularios Windows

A pesar de que el apelativo *.NET* hace que asociamos rápidamente Visual Basic .NET con el desarrollo de aplicaciones y servicios para Internet e *intranets* corporativas, lo cierto es que aún hoy, comenzando el año 2002, una gran parte de los proyectos que se desarrollan tienen como destinatario el sistema operativo Windows en alguna de sus diferentes versiones: 98, Millenium, NT, 2000 o bien XP.

No es de extrañar ya que Windows es el sistema operativo dominante en los sistemas de escritorio, los que utiliza el usuario final, y en una parte importante de servidores de aplicaciones y datos.

Como no podía ser de otra manera, Visual Basic .NET incorpora todos los elementos necesarios para desarrollar aplicaciones para Windows. En realidad, con Visual Basic .NET no podemos crear aplicaciones nativas para dicho sistema, sino aplicaciones que se ejecutan en Windows con la plataforma .NET instalada.

Hasta ahora no hay ninguna versión de Windows que incorpore dicha plataforma, la primera será Windows .NET, pero posiblemente aparezca como un *Service Pack* o actualización para Windows XP, 2000 y otras versiones más antiguas.

Teniendo instalada la plataforma .NET, nuestras aplicaciones Windows, desarrolladas con Visual Basic .NET, podrán hacer uso de los formularios Windows, GDI+ y otros servicios para Windows existentes en .NET. En este capítulo nos centraremos en el uso de formularios Windows para crear interfaces de usuario Windows.

¿Qué son los formularios Windows?

Como ya sabe, se indicó en el tercer capítulo, la plataforma .NET incorpora una serie de servicios accesibles para las aplicaciones que podemos desarrollar con Visual Basic .NET. En la figura 3.8 se representaban algunos de esos servicios, parte de los cuales agrupábamos bajo la denominación *Win Forms*.

La plataforma .NET incorpora un ámbito, `System.Windows.Forms` concretamente, en el que existen múltiples clases cuya finalidad es facilitar el desarrollo de aplicaciones Windows. Esas clases pueden usarse desde cualquier lenguaje .NET. Visual Studio .NET, además, incorpora asistentes y diseñadores capaces de generar y administrar el código de este tipo de aplicaciones en dos lenguajes: Visual Basic .NET, que es el que nos interesa de forma particular, y Visual C#.

Una de las clases existentes en dicho ámbito es `Form`, clase que representa al típico formulario que estábamos acostumbrados a usar en versiones previas de Visual Basic. La diferencia es que en dichas versiones el formulario era casi algo mágico, un elemento que aparecía en el proyecto y sobre el que podíamos insertar componentes. En Visual Basic .NET, por el contrario, todo formulario diseñado por nosotros define automáticamente, como podremos ver, una clase derivada `Form`, clase sobre la cual tenemos un control absoluto.

En Visual Basic .NET, por tanto, un formulario Windows es un objeto de la clase `System.Windows.Forms.Form` o de una clase derivada de ésta. Como veremos de inmediato, los asistentes y diseñadores visuales se encargan de generar la clase derivada e ir personalizándola a medida que se insertan componentes, modifican propiedades y gestionan eventos. Nuestro trabajo, como en versiones previas de Visual Basic, se limitará a operaciones de arrastrar y soltar, edición de propiedades en una ventana y escritura del código importante, la lógica de negocio, asociada a los eventos.

Una aplicación Windows sencilla

Para estudiar los elementos que forman parte de una aplicación Windows comenzaremos creando una bastante sencilla, utilizando el diseñador de formularios y elementos como la ventana Propiedades y el Cuadro de herramientas, para después analizar el código resultante.

Iniciamos un nuevo proyecto Aplicación para Windows con Visual Basic .NET, como se aprecia en la figura 10.1. Obtenemos un proyecto formado por dos módulos: `AssemblyInfo.vb` y `Form1.vb`. El primero de ellos aloja información sobre el ensamblado que generará el proyecto, mientras que el segundo contiene el código que creará y gestionará el formulario.

Lo que se ve en la parte central del entorno es una ventana vacía, un formulario. Éste es representación del código contenido actualmente en el módulo `Form1.vb`. De hecho, para alterar las características y comportamiento de este formulario tenemos dos posibilidades: editar directamente ese código o utilizar el diseñador de formularios.

Figura 10.1. Seleccionamos el tipo de proyecto

Diseño del formulario

Utilizando algunos componentes del **Cuadro de herramientas** y la ventana **Propiedades** dejaremos el formulario tal y como puede verse en la figura 10.2. Hemos insertado los siguientes componentes: un `Label`, un `TextBox`, un `Button` y un `ListBox`. Editando la propiedad `Text` de los tres primeros modificaremos los titulos y contenidos. También alteramos la propiedad `Name` del `TextBox` y el `ListBox` para darles un nombre más lógico que los establecidos por defecto.

La caja de texto, nombre con el que se conoce de forma habitual al control `TextBox`, nos servirá para introducir datos que irán añadiéndose a la lista, dispuesta debajo, cada vez que se pulse el botón titulado **Añadir**.

Lógicamente podriamos modificar muchas otras propiedades, tanto del formulario como de los controles. Tan sólo tendriamos que conocer el significado de cada una de ellas y los valores que pueden tomar. Ésta es una información de referencia que puede encontrar en la propia ayuda de Visual Basic .NET y no tiene sentido enumerarla aquí propiedad por propiedad, método por método y evento por evento.

Implementación de funcionalidad

Si ejecutamos ahora mismo nuestro proyecto veremos aparecer la ventana del programa, podremos pulsar el botón que hay en ella e, incluso, introducir un dato en la caja de texto. No veremos, sin embargo, ninguna reacción por parte del programa, aparte de cerrar la ventana al pulsar el botón correspondiente.

Para que al pulsar el botón **Añadir** el dato introducido en la caja de texto se añada a la lista, tal y como deseamos, tendremos que escribir algo de código. Este código deberá ejecutarse cuando se produzca el evento `Click` del mencionado botón. Tenemos dos opciones: abrir la lista de eventos del componente

que aparece en la parte superior del editor de código y seleccionar el evento `Click`, sería el procedimiento habitual para la mayoría de los eventos, o hacer doble clic directamente sobre el botón, ya que el evento por defecto es `Click`.

El diseñador genera un nuevo método y nosotros tan sólo tenemos que introducir en él las tres sentencias siguientes:

```
lbDatos.Items.Add(tbDato.Text)
tbDato.SelectAll()
tbDato.Focus()
```

La propiedad `Items` del `ListBox` nos permite acceder a la lista de elementos que contiene este control. Con el método `Add()`, como puede suponer, añadimos un elemento a esa lista, concretamente el texto introducido en el `TextBox`. Éste, el control llamado `tbDato`, mantendrá el texto introducido, texto que seleccionamos mediante el método `SelectAll()` para así facilitar la introducción de otro dato. Finalmente, utilizamos el método `Focus()` para devolver el foco de entrada a la caja de texto ya que, tras pulsar el botón, el foco de entrada estaría en otro punto.

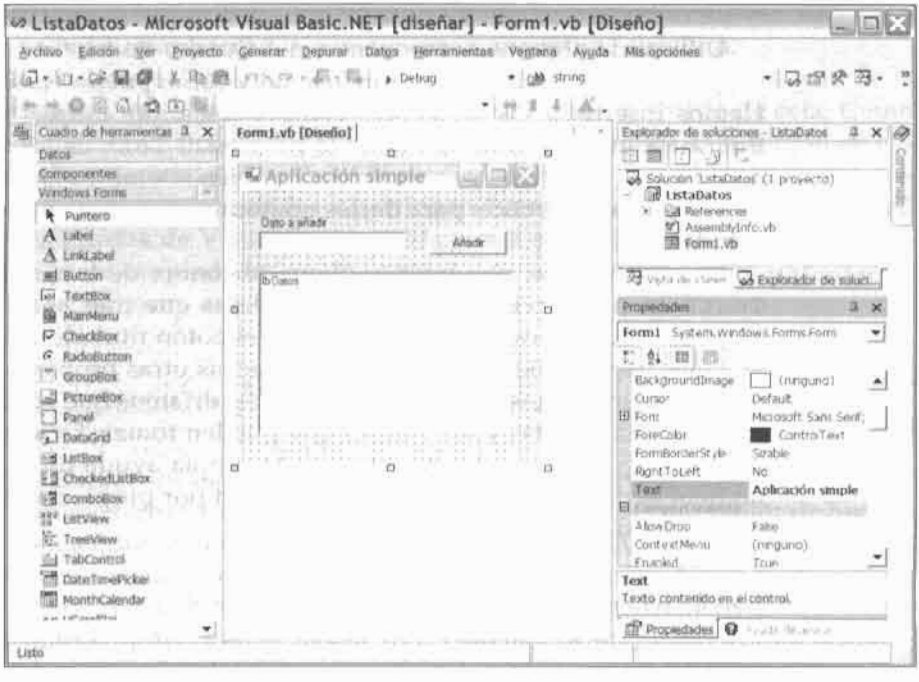


Figura 10.2. Aspecto del formulario tras introducir los controles y modificar propiedades

Análisis del código generado

El programa se comporta ahora como deseábamos. En la figura 10.3 puede ver su apariencia tras añadir algunos datos en la lista. Nosotros tan sólo hemos

escrito el código necesario para añadir cada dato introducido a la lista, el resto del código ha sido generado automáticamente.

Es necesario analizarlo para comprender qué es lo que está ocurriendo, por qué aparece la ventana, los controles se colocan en la posición que habíamos establecido en el diseñador, etc.

Figura 10.3. Aspecto del programa en funcionamiento

Vamos a ir recorriendo el módulo de código de principio a fin, seccionándolo y comentándolo para saber qué es lo que hace funcionar nuestra aplicación. Lo primero que encontramos es la definición de una nueva clase, llamada `Form1`, que es derivada de la clase `Form`. Ésta se encuentra definida en el ámbito `System.Windows.Forms`:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
```

Puede utilizar el Examinador de objetos para ver cuál es el contenido de este ámbito y los enumerados en la lista Importaciones del proyecto (véase la figura 10.4). Esta lista puede usarse para importar ámbitos con nombre sin necesidad de utilizar la sentencia `Imports`. El que más nos interesa en este momento es `System.Windows.Forms`, en el que se alojan las clases correspondientes al formulario y los controles.

Nota

Aunque la clase `Form` se encuentra en `System.Windows.Forms`, tal y como se ha indicado, la clase `Form1`, definida automáticamente por el diseñador, se alojará en el ámbito de nuestro proyecto que, como recordará, se establece en la ventana de propiedades en la página **General**.

Al estar derivada de `System.Windows.Forms.Form`, la clase `Form1` ya cuenta con una serie de propiedades y características que le hacen comportarse como una ventana típica de Windows. Ésta es, precisamente, una de las ventajas de la herencia.

Figura 10.4. Ámbitos con nombre importados por el proyecto

Continuando con nuestro recorrido por el módulo de código, lo siguiente que encontramos es una etiqueta con el título Windows Form Designer generated code, tal como se aprecia en la parte superior de la figura 10.5. Puede situar el puntero del ratón sobre dicha etiqueta para ver, en una ventana emergente, parte del código que contiene esa región. Para acceder completamente al código, inspeccionándolo, tan sólo tenemos que pulsar sobre el signo + que aparece en el margen izquierdo a la altura de la citada etiqueta.

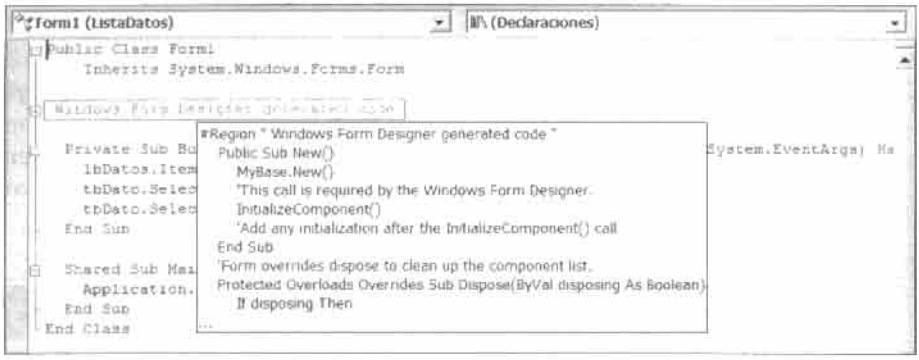


Figura 10.5. Parte del código está oculto en una región

Lo primero que encontramos en esta región es el constructor de la clase, encargado de configurar el formulario antes de que éste se haga visible en pantalla. Como puede verse a continuación, en principio el constructor sólo contiene una llamada al constructor base y otra al método `InitializeComponent()`, sobre el que volveremos después, y una serie de comentarios. Es el punto adecuado para que añadamos manualmente cualquier código de preparación que necesitemos.

```
Public Sub New()
    MyBase.New()

    InitializeComponent()

End Sub
```

El código del destructor, dispuesto a continuación del constructor, se encarga de comprobar si la variable `components` contiene alguna referencia, caso en el cual se procede a la correspondiente liberación de recursos. Después se llama al método `Dispose()` de la clase base, completando así el proceso de destrucción.

```
Protected Overloads Overrides Sub Dispose( _
    ByVal disposing As Boolean)
     If disposing Then
         If Not (components Is Nothing) Then
             components.Dispose()
         End If
     End If
     MyBase.Dispose(disposing)
End Sub
```

Tras el constructor y el destructor encontramos la definición de cuatro miembros de datos, concretamente cuatro objetos de otras tantas clases:

```
Friend WithEvents Label1 As _
      System.Windows.Forms.Label
Friend WithEvents tbDato As _
      System.Windows.Forms.TextBox
Friend WithEvents Button1 As _
      System.Windows.Forms.Button
Friend WithEvents lbDatos As _
      System.Windows.Forms.ListBox
```

Cada una de estas variables representa a uno de los controles que hemos insertado previamente en el formulario. Ha sido el diseñador el que, cuando hemos insertado cada control, se ha ocupado de escribir este código, aparte del necesario para crear cada objeto e insertarlo en el contenedor que es el formulario, como veremos en un momento.

Nota

Ya que previamente se ha importado el ámbito `System.Windows.Forms`, para hacer referencia a las clases `Label`, `TextBox`, `ListBox` y `Button` no sería preciso utilizar una referencia completa y nos bastaría con usar el

nombre de la clase. El diseñador, sin embargo, genera estas referencias completas para evitar posibles conflictos con otras clases que, habiendo sido definidas en otros ámbitos, coincidan en nombre.

Observe el uso de los modificadores `Friend` y `WithEvents`. El primero establece el ámbito de visibilidad de los identificadores de tal manera que puedan ser accesibles por clases derivadas y desde cualquier punto del módulo actual. El segundo prepara el objeto para que sus eventos puedan ser interceptados por métodos de nuestra clase, según se vio en un capítulo anterior.

Aparte de esas cuatro variables, justo después encontramos otra más:

```
Private components As _
   System.ComponentModel.Container
```

La variable `components`, en principio con un valor nulo, contendrá una referencia a un objeto de la clase `Container`. Esta clase se utiliza, en este caso concreto, para contener los controles que aparecerán en el formulario y es usada internamente por el diseñador de formularios, por lo que no debemos manipularla si pretendemos seguir trabajando de manera visual sobre la ventana.

Lo siguiente que encontramos es el método `InitializeComponent()` al que se llama desde el constructor del formulario. Su finalidad es traducir todas las acciones que hemos realizado en el diseñador, insertando componentes y personalizando sus propiedades, en sentencias que, al ser ejecutadas, reproducirán el aspecto de la ventana tal y como la hemos diseñado. El código de este método es el siguiente:

```
Private Sub InitializeComponent()
  Me.Label1 = New System.Windows.Forms.Label()
  Me.tbDato = New System.Windows.Forms.TextBox()
  Me.Button1 = New System.Windows.Forms.Button()
  Me.lbDatos = New System.Windows.Forms.ListBox()
  Me.SuspendLayout()

  Me.Label1.AutoSize = True
  Me.Label1.Location = New _
            System.Drawing.Point(16, 24)
  Me.Label1.Name = "Label1"
  Me.Label1.Size = New System.Drawing.Size(72, 13)
  Me.Label1.TabIndex = 0
  Me.Label1.Text = "Dato a añadir"

  Me.tbDato.Location = New _
            System.Drawing.Point(16, 40)
  Me.tbDato.Name = "tbDato"
  Me.tbDato.Size = New System.Drawing.Size(144, 20)
  Me.tbDato.TabIndex = 1
  Me.tbDato.Text = ""
```

```
    Me.Button1.Location = New _
            System.Drawing.Point(184, 40)
    Me.Button1.Name = "Button1"
    Me.Button1.TabIndex = 2
    Me.Button1.Text = "Añadir"



    Me.lbDatos.Location = New _
            System.Drawing.Point(16, 80)
    Me.lbDatos.Name = "lbDatos"
    Me.lbDatos.Size = New _
            System.Drawing.Size(248, 160)
    Me.lbDatos.TabIndex = 3



    Me.AutoScaleBaseSize = New _
            System.Drawing.Size(5, 13)
    Me.ClientSize = New _
            System.Drawing.Size(292, 266)
    Me.Controls.AddRange(New _
            System.Windows.Forms.Control() _
            {Me.lbDatos, Me.Button1, _
             Me.tbDato, Me.Label1})
    Me.Name = "Form1"
    Me.Text = "Aplicacion simple"
    Me.ResumeLayout(False)

  End Sub
```

Las primeras cuatro sentencias que encontramos son las encargadas de crear los objetos de las clases `Label`, `TextBox`, `Button` y `ListBox`, guardando las respectivas referencias en las variables que, a tal efecto, se habían declarado al principio de la clase.

A continuación encontramos cuatro bloques de varias sentencias, cada uno de ellos encargado de configurar cada uno de los controles. Se establece su nombre, título, orden de acceso mediante el tabulador, posición y dimensiones. Observe que las propiedades `Location` y `Size` toman como valor objetos de las clases `Point` y `Size`, definidas en el ámbito `System.Drawing`.

Finalmente, en lo relativo al método `InitializeComponent()`, encontramos el bloque de sentencias que configuran el formulario. Las primeras establecen sus dimensiones. A continuación tenemos otra que llama al método `AddRange()` de la propiedad `Controls` para añadir los cuatro controles anteriormente creados. La propiedad `Controls` da acceso a la lista de componentes contenidos en otro, en este caso los controles contenidos en el formulario. El método `AddRange()` añade a esa lista los componentes que se facilitan como parámetro en forma de arreglo. En este caso concreto, el arreglo se define en la propia llamada al método.

Por último tenemos un método al que corresponde la siguiente cabecera:

```
Private Sub Button1_Click( _
  ByVal sender As System.Object, _
  ByVal e As System.EventArgs) _
  Handles Button1.Click
```

Observe cómo se utiliza la palabra clave `Handles` para asociar la ejecución de este método con el evento `Click` de `Button1`. En un capítulo previo vimos otra forma de hacer lo mismo con `AddHandler`.

El punto de entrada a este programa no es el típico método `Main()`, como en los ejemplos de capítulos previos, sino el propio formulario. Abra la ventana de propiedades del proyecto y compruebe el valor de la lista Objeto inicial. Verá, como en la figura 10.6, que se apunta al objeto `Form1`.

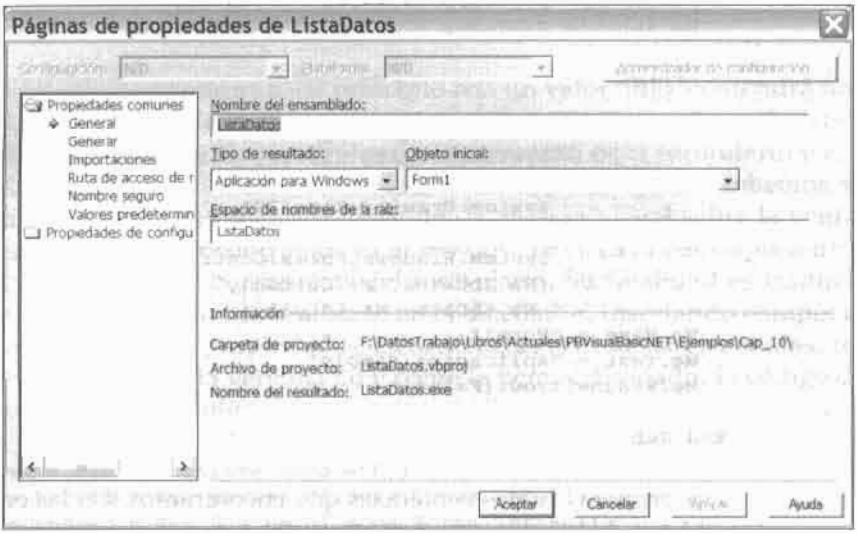


Figura 10.6. Objeto que será creado para poner en marcha la aplicación

Resumiendo

Este programa simple pone en marcha una aplicación Windows, algo que podríamos hacer también llamando al método `Run()` de la clase `Application` desde nuestro método `Main()`.

Opcionalmente podemos facilitar un objeto de una clase derivada de `Form`, un formulario, que se haría visible tras iniciarse la aplicación. Lógicamente, en el módulo podríamos tener tantas definiciones de formularios como precisemos, aunque lo habitual es que cada formulario, y sus miembros, se alojen en un módulo independiente.

Añada al final de la definición de la clase `Form1` el código siguiente:

```
Shared Sub Main()
  Application.Run(New Form1())
End Sub
```

A continuación abra la ventana de propiedades del proyecto e indique que el punto de entrada es el método `Main()` y ejecute el proyecto. El resultado, como podrá ver, es idéntico, pero en este caso somos nosotros los que ponemos en marcha la aplicación y creamos el formulario.

Obviamente, podríamos generar esta misma aplicación escribiendo nosotros mismos el código. También es obvio que resulta mucho más fácil conseguir el mismo resultado utilizando el diseñador de formularios, el Cuadro de herramientas y la ventana Propiedades, dejando que sean ellos los que generen y manipulen el código. No es habitual, al menos trabajando con Visual Basic .NET, que codifiquemos tareas como la creación de los controles y el establecimiento de su posición y dimensiones.

Lo único que necesitamos, por tanto, es conocer los elementos que tenemos disponibles en el Cuadro de herramientas, saber cómo instalar cualquier otro control que pudiéramos necesitar y, lógicamente, contar con información de las propiedades, métodos y eventos de cada uno de esos componentes.

Información sobre la aplicación

Como acabamos de ver, una aplicación Windows se pone en marcha automáticamente creando un objeto de una clase derivada de `Form`. En realidad, tras el código que nosotros podemos ver se utilizan los métodos de la clase `Application`. No es preciso crear un objeto de esta clase ya que la mayoría de sus miembros son estáticos, lo cual significa que pueden ser usados de manera directa, sin estar asociados a un objeto. Esta clase dispone de propiedades con datos informativos diversos, así como métodos que controlan el flujo de la aplicación. También genera algunos eventos interesantes.

Inicio de la aplicación

Cuando ejecutamos una aplicación el control suele tomarlo un método compartido definido en alguna de las clases, típicamente el método `Main()`. Cuando desde ese método se llama al método `Run()` de la clase `Application` lo que se hace es poner en manos de dicha clase el control de la aplicación, dejando que sea ella la que procese los mensajes y se encargue de gestionarlos. Este proceso es efectuado automáticamente por Visual Basic .NET cuando no se codifica explícitamente el método `Main()` y, en su lugar, se indica como punto de entrada a la aplicación la clase del formulario.

Podemos llamar al método `Run()` de `Application` sin facilitar parámetro alguno, caso en el cual se pone en marcha el bucle de proceso de mensajes sin mostrar ninguna ventana. La segunda alternativa es la que hemos usado en el ejemplo anterior, entregando como parámetro una referencia al formulario que desea mostrarse como ventana inicial o principal.

Otra posibilidad, a la hora de invocar al método `Run()`, consiste en crear un objeto de la clase `ApplicationContext` y asignar a la propiedad `MainForm` una referencia al formulario a mostrar inicialmente.

En cualquier caso, el objeto `Application` se encargará, como se ha dicho, de procesar los mensajes que reciba del sistema o bien remitirlos al destinatario que corresponda, por ejemplo el botón o la caja de texto, para que los procese adecuadamente.

Datos del entorno de la aplicación

Al iniciarse la aplicación, llamando al método `Run()`, automáticamente se recuperan una serie de parámetros que son alojados en el interior de los miembros de la clase `Application`. Podemos recuperar esos datos mediante múltiples propiedades. En la tabla 10.1 se resumen algunas de ellas indicando la información que contienen.

Tabla 10.1. Propiedades informativas de la clase `Application`

Propiedad	Contenido
`StartupPath`	Camino desde el que se ha iniciado la aplicación
`ExecutablePath`	Camino y nombre del ejecutable en el que se aloja la aplicación
`CompanyName`	Nombre de la empresa que ha desarrollado la aplicación
`ProductName`	Nombre de la aplicación
`ProductVersion`	Versión de la aplicación
`UserAppDataPath`	Camino a la carpeta donde alojar información de usuario
`CommonAppDataPath`	Camino a la carpeta donde alojar información general a todos los usuarios

Podemos preparar una aplicación sencilla que muestre estos datos en una lista a efectos de comprobación. Insertamos en un formulario un control `ListBox`, hacemos doble clic sobre el fondo del formulario para abrir el método correspondiente al evento `Load` e introducimos el código siguiente:

```
Dim Datos() = { _
"StartupPath= " + Application.StartupPath, _
"ExecutablePath= " + _
      Application.ExecutablePath, _
"CompanyName= " + Application.CompanyName, _
"ProductName= " + Application.ProductName, _
"ProductVersion= " + _
      Application.ProductVersion, _
"UserAppDataPath= " + _
      Application.UserAppDataPath, _
"CommonAppDataPath= " + _
Application.CommonAppDataPath() _
}

Dim Indice As Byte
```

```
For Indice = 0 To Datos.GetUpperBound(0) - 1
  lbParametros.Items.Add(Datos(Indice))
Next
```

Preparamos, en un arreglo de cadenas, los datos que deseamos mostrar. A continuación la recorremos en un bucle añadiendo cada una de ellas a la lista. El resultado es el que puede verse en la figura 10.7.

Figura 10.7. El programa mostrando diversos datos del entorno

Nota

El dato alojado en `ProductVersion` se actualiza automáticamente cada vez que se compila el proyecto, consiguiendo así la diferenciación de diferentes versiones de la aplicación aún cuando sus índices de mayor peso no cambien.

Otros miembros de interés

Aparte de propiedades, en su mayor parte informativas, la clase `Application` también dispone de algunos métodos y eventos que pueden resultar interesantes.

Para poner fin al bucle de proceso de mensajes, en la práctica finalizar la ejecución del programa y cerrar sus ventanas, podemos llamar al método `Exit()`. El control se devolverá a la sentencia siguiente a la que llamó a `Run()`, generalmente en el método `Main()`.

Mediante el método `DoEvents()` habilitaremos el proceso de los mensajes que haya pendientes. Esto, normalmente, no es preciso hacerlo de manera explícita a menos que estemos ejecutando algún trabajo lento, que requiera mucho tiempo. La alternativa, como ya sabemos, es ejecutar ese trabajo lento en un hilo concurrente.

Los cuatro eventos de que dispone la clase `Application` son `ApplicationExit`, `Idle`, `ThreadException` y `ThreadExit`. El primero de ellos se

genera justo antes de que la aplicación termine, el segundo cuando la aplicación ha terminado un trabajo y va a quedar en espera, el tercero cuando se produce alguna excepción y, finalmente, el último cada vez que finaliza un hilo de ejecución o *thread*, incluido el principal.

Aplicaciones de documento múltiple

Como hemos visto, en el ejemplo anterior, cuando una aplicación Windows se pone en marcha automáticamente aparece la ventana inicial, establecida por nosotros mismos al llamar al método `Run()` o bien en la ventana de propiedades del proyecto.

Lógicamente, la aplicación puede necesitar múltiples ventanas más. Éstas deberán ser abiertas desde la ventana inicial a demanda del usuario. Es lo que ocurre habitualmente, por ejemplo, con los cuadros de diálogo.

El formulario o ventana principal de la aplicación puede actuar como contenedor de otras ventanas, dando lugar a la típica aplicación MDI o de documento múltiple. También tenemos a nuestra disposición algunos cuadros de diálogo prefabricados.

Vamos a conocer algunos conceptos relativos al trabajo con ventanas basándonos en un ejemplo cuyo resultado final será el mostrado en la figura 10.8: una aplicación MDI, similar a un editor multiarchivo, en el que podremos elegir el color del texto de cada ventana.

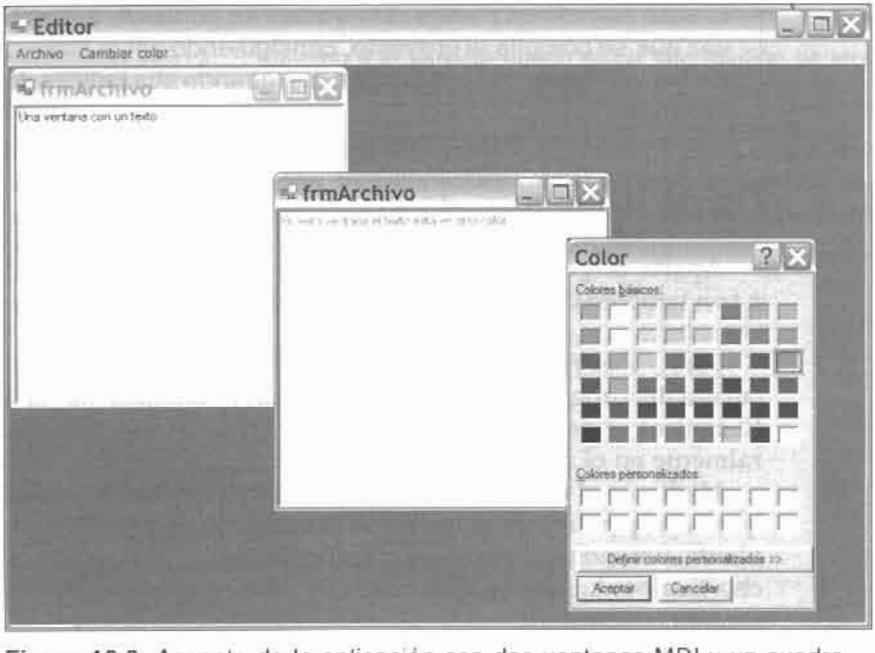


Figura 10.8. *Aspecto de la aplicación con dos ventanas MDI y un cuadro de diálogo abiertos*

Preparación de la ventana principal

Asumamos que acabamos de iniciar un nuevo proyecto, tal y como hemos hecho en los dos ejemplos previos, por lo que disponemos, en este momento, de un formulario vacío que se comporta como una ventana estándar. Para convertir esa ventana en un contenedor MDI, capaz de alojar otros formularios, lo único que hemos de hacer es dar el valor `True` a la propiedad `IsMDIContainer`. Al hacerlo verá que el fondo de la ventana cambia de color.

Si examina el código del formulario, tras modificar esa propiedad, observará que lo que hemos hecho es, realmente, algo más complejo. No sólo se asigna el valor `True` a la propiedad `IsMDIContainer` del formulario sino que, además, se crea un objeto de la clase `MdiClient`. Este control es el fondo más oscuro que vemos en el interior del formulario y que ocupa todo el área disponible. Como cualquier otro control, es añadido al formulario mediante el método `AddRange()` de la propiedad `Controls`.

Una ventana marco MDI puede contener otras en su interior, concretamente ventanas que tengan el valor `True` en su propiedad `IsMDIChild` y cuya propiedad `MDIParent` contenga una referencia a la ventana marco.

La clase `MdiClient` cuenta con una propiedad `MdiChildren`, que facilita una lista de todas las ventanas hija abiertas, y otra llamada `BackgroundImage`, mediante la cual podemos establecer una imagen o gráfico de fondo en la ventana. Estas dos propiedades aparecen también en la clase `Form`, de tal manera que no es necesario usar explícitamente el objeto `MdiClient`.

Creación de un menú de opciones

El siguiente paso que daremos será crear un menú de opciones, realmente con una sola opción, que nos servirá para poder abrir ventanas hija cada vez que lo necesitemos. Tomamos un componente `MainMenu` del Cuadro de herramientas y lo insertamos en el formulario. Realmente aparecerá en la parte inferior del diseñador, mientras que en el formulario todo está preparado para que introduzcamos los títulos de las opciones.

Introducimos como nombre de la opción principal Archivo y, tras desplazarnos abajo, facilitamos el título de la única opción: Nuevo. En la figura 10.9 puede ver el aspecto del menú durante la fase de diseño. Este es un aspecto que se ha mejorado sustancialmente respecto a versiones previas de Visual Basic, en las que el diseño de un menú resultaba bastante menos intuitivo.

Lógicamente, la opción Archivo>Nuevo tendrá un código asociado ya que, de lo contrario, de poco serviría. Hacemos doble clic sobre la opción, causando la generación del código necesario para gestionar el evento `Click`. En su interior introducimos las siguientes cuatro sentencias:

```
Dim VentanaHija As frmarchivo = New frmArchivo()

VentanaHija.MdiParent = Me
VentanaHija.Show()
```

Figura 10.9. Diseñamos el menú principal del formulario

Lo primero que hacemos es crear un objeto de la clase `frmArchivo`, una clase que, ciertamente, aún no hemos definido pero de la que nos ocuparemos de inmediato. Ese objeto será una ventana, un formulario, de la cual modificamos una propiedad: `MdiParent`. Con ella enlazamos esta nueva ventana con la que actuará como padre de ambas ventanas. Recuerde que en este contexto `Me` representa al formulario principal.

Finalmente utilizamos el método `Show()` para hacer aparecer la ventana. Este mismo método podríamos usarlo para mostrar cualquier ventana, aunque no fuese MDI. La ventana aparece como no modal, lo cual significa que puede accederse a los otros elementos de la aplicación sin necesidad de cerrarla. Para mostrar una ventana modal usaríamos el método `ShowModal()`.

Diseño de la ventana hija MDI

Si compilásemos el proyecto tal y como está ahora obtendríamos algunos errores, ya que la clase `frmArchivo`, a la que se hace referencia desde el código anterior, no existe actualmente. Vamos a crearla añadiendo un nuevo elemento al proyecto, concretamente un nuevo formulario. Para ello utilizaremos la opción Agregar nuevo elemento, como puede verse en la figura 10.10, seleccionando el elemento Windows Form.

Añadiremos a este formulario tres componentes: un `MainMenu`, un `ColorDialog` y un `TextBox`. De éste último modificaremos la propiedad `Multiline`, asignándole el valor `True`, y la propiedad `Dock`. Ésta cuenta con un editor específico que, al ser desplegado (véase figura 10.11) nos permite seleccionar el área al que se adosará el control. En este caso seleccionamos el botón central, correspondiente al valor `Fill`, de tal forma que el `TextBox` ocupe todo el espacio disponible en la ventana.

Utilizaremos el componente `MainMenu` para incorporar una opción, a la que llamaremos Cambiar color, cuya finalidad será abrir un cuadro de diálogo que permita al usuario seleccionar el color en el que desea el texto. Este cuadro de diálogo, como puede suponer, es el ofrecido por el componente `ColorDialog`. Hacemos doble clic sobre la opción, una vez añadida, e introducimos el código siguiente en el método generado:

```
If ColorDialog1.ShowDialog() =
                DialogResult.OK Then

  TextBox1.ForeColor = ColorDialog1.Color
End If
```

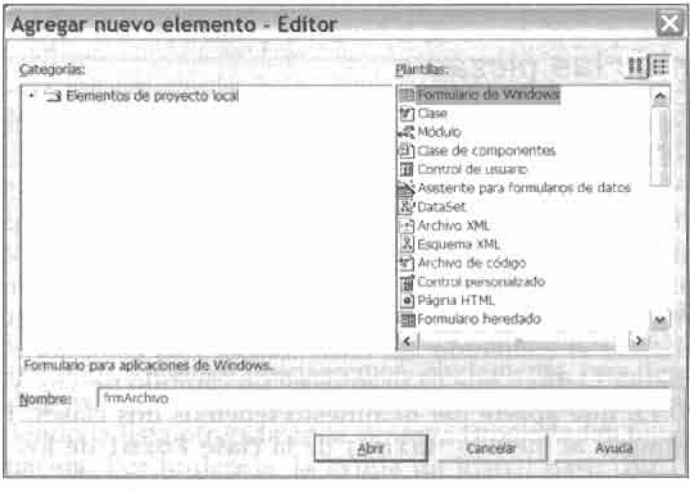


Figura 10.10. Añadimos un nuevo formulario al proyecto

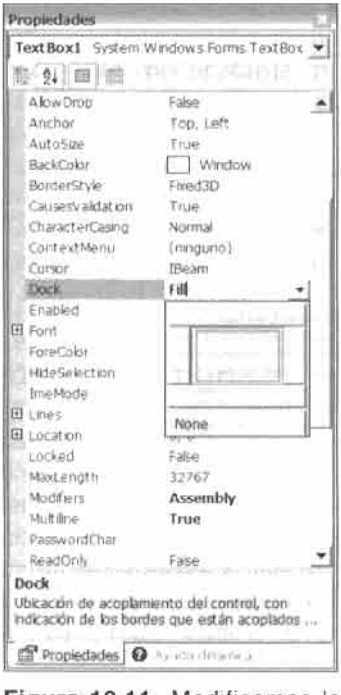


Figura 10.11. Modificamos la propiedad `Dock` del componente `TextBox`

Llamamos al método `ShowDialog()` del componente `ColorDialog` para mostrar el cuadro de diálogo. Si dicho método devuelve el valor `DialogResult.OK` es que se ha seleccionado un color y cerrado la ventana pulsando el botón **OK**, caso en el cual tomamos el color elegido, de la propiedad `Color`, y lo usamos para modificar la propiedad `ForeColor` del `TextBox`. Es decir, cambiamos el color del texto que haya en la caja de texto.

Unión de las piezas

Nuestro proyecto consta, en este momento, de dos módulos de código, cada uno de ellos correspondiente a un formulario. Si examina el código verá que ambos forman parte del mismo ámbito con nombre. De hecho, al compilar se generará un solo ensamblado. Por ello desde el formulario principal podemos crear un objeto de la clase `frmArchivo`, el formulario hijo MDI, sin necesidad de utilizar archivos de cabecera ni establecer ninguna referencia. El desarrollo, por tanto, resulta mucho más simple que usando otros lenguajes y entornos.

Puede ver el contenido del ensamblado, tras compilar el proyecto, con la herramienta `ildasm` que ya usamos en un capítulo previo. Verá, como en la figura 10.12, que aparte del manifiesto tenemos dos clases. Fíjese en la marca que distingue al método `Main()` de la clase `Form1` de los demás, indicando que es estático (compartido). Ese método ha sido creado automáticamente por Visual Basic y no lo vemos en el editor de código.

Figura 10.12. Contenido del ensamblado obtenido a partir del proyecto

Al ejecutar el programa se encontrará, en principio, con la ventana marco MDI vacía. Puede usar la única opción disponible para abrir una o más ventanas de edición. Observe que el menú que habíamos definido en la ventana hija se integra con el principal, apareciendo en la ventana marco.

Trabajo con componentes

El trabajo con ventanas, ya sean de tipo MDI o cuadros de diálogo, no comporta mucha más complicación. Lo único que necesitamos, para construir cualquier aplicación Windows con interfaz de usuario, es conocer los controles que aparecen en la página Windows Forms del Cuadro de herramientas. En su mayor parte son equivalentes a los controles que podíamos encontrar en Visual Basic 6, aunque con diferencias puntuales.

Lo único que necesitamos, por tanto, es saber cuál es la finalidad de cada componente, algo que, generalmente, es fácil deducir del nombre e icono que le representan. También precisaremos, obviamente, una descripción de las propiedades, métodos y eventos con que cuenta cada componente. Simplemente recorriendo la lista ofrecida en la ventana Propiedades tendremos parte de esa información. Por lo demás, la ayuda de Visual Basic .NET contiene información de referencia de cada componente y cada uno de sus miembros.

Desde una aplicación Windows, aparte de contar con una interfaz de usuario para solicitar y mostrar información, suelen necesitarse otros elementos como componentes para el acceso a datos o a reglas que se ejecutan remotamente. Esta necesidad, no obstante, no es exclusiva de las aplicaciones basadas en *Windows Forms* y, por ello, las abordaremos de una forma más general en capítulos posteriores.

Aparte de los controles inicialmente visibles en el Cuadro de herramientas, podemos usar la opción de personalización, en el menú emergente, para acceder a la ventana mostrada en la figura 10.13. Desde ella podemos instalar otros componentes, tanto componentes .NET como componentes COM, que estén registrados en el sistema.

Miembros comunes

Aunque, como acaba de indicarse, podemos instalar tantos nuevos componentes como sean necesarios, lo cierto es que en la mayoría de nuestros diseños de interfaz de usuario utilizaremos siempre un conjunto limitado de controles: botones, cajas de texto, listas, etiquetas, cajas de selección, botones de radio y cuadrículas de datos son algunos de ellos. Estos controles están representados por las clases `Button`, `TextBox`, `ListBox`, `Label`, `CheckBox`, `RadioButton` y `DataGrid`, todas ellas derivadas, directa o indirectamente, de la clase `Control` que, a su vez, es una derivada de `Component`. Esta raíz común, como podrá suponer, aporta una serie de miembros comunes a todos esos componentes, miembros que podemos conocer de manera general.

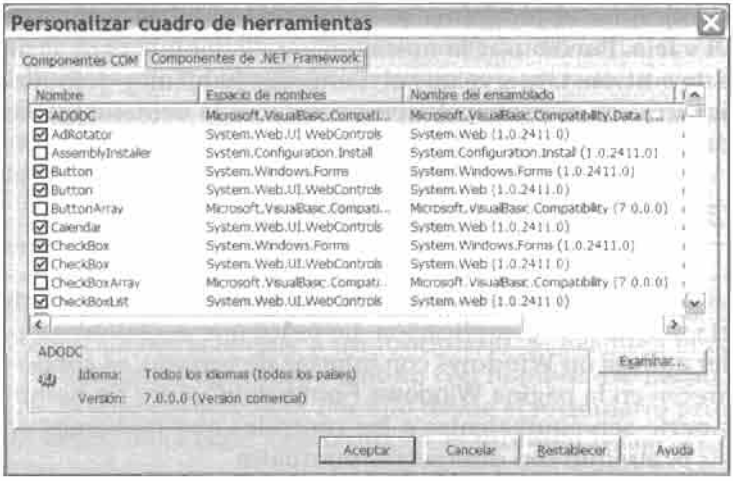


Figura 10.13. Añadimos nuevos componentes al Cuadro de herramientas

La clase `Control` se encarga de gestionar la posición y las dimensiones del control, procesar la introducción de información a través del teclado y el ratón, controlar el anclaje y adosado en el interior de otros controles o administrar el foco de entrada y el orden de acceso. Todas estas tareas, por tanto, tan sólo tendremos que abordarlas una vez y, posteriormente, aplicarla a cualquiera de las clases de control derivadas de `Control`.

Posición y dimensiones

Todos los controles mencionados se diseñan para ser insertados en un contenedor, concretamente en un formulario Windows. Por ello cuentan con propiedades que mantienen la posición del control dentro del contenedor, así como las dimensiones. Son datos que se establecen en puntos de pantalla, como es habitual en la mayoría de las herramientas de desarrollo, y no en *twips* como ocurría en Visual Basic 6.

La posición se aloja en las propiedades `Left` y `Top`, que representan el punto horizontal y vertical, respectivamente, de la esquina superior izquierda donde aparecerá el control. `Left` y `Top` son de tipo `Integer`, es decir, contienen directamente el punto horizontal y vertical. También la propiedad `Location` mantiene esa misma información, en este caso en forma de una estructura `Point`. Dicha estructura cuenta con dos propiedades, `X` e `Y`, que mantienen el punto horizontal y vertical, así como una serie de métodos y de operadores que permiten operar sobre ese punto (X,Y) para trasladarlo en el plano, compararlo con otro punto, etc.

El tamaño del control también podemos obtenerlo de distintas propiedades. Por una parte están `Height` y `Width` que, al igual que `Left` y `Top`, son de tipo `Integer`. Contienen el número de puntos del alto y ancho, respectivamente, que tiene el control. Si no nos interesan las dimensiones sino el punto de la esquina opuesta del control, la esquina inferior derecha, nos serviremos de las propiedades `Right` y `Bottom`.

También podemos recuperar toda la información relativa a posición y dimensiones leyendo la propiedad `Bounds`, de tipo `Rectangle`. La estructura `Rectangle` dispone de cuatro propiedades: `X`, `Y`, `Width` y `Height`, conteniendo posición horizontal, posición vertical, ancho y alto del control. Al igual que `Point`, la estructura `Rectangle` dispone de una serie de métodos que facilitan ciertas operaciones con áreas rectangulares, como: aumento o reducción de dimensiones, unión, intersección, etc. Una propiedad similar a `Bounds` es `ClientRectangle`, si bien en este caso la estructura contendrá el valor `0` en `X` e `Y` y las dimensiones en `Width` y `Height`.

Nota

Las propiedades citadas no son meramente informativas. Podemos modificar el valor de cualquiera de ellas para alterar la posición o el tamaño del control que corresponda.

Por último, en lo relativo a la posición y dimensiones de los controles, citar los métodos `SetBounds()`, `SetLocation()` y `SetClientArea()`, mediante los cuales podemos establecer las coordenadas del control, su posición o su tamaño, respectivamente.

Orden de acceso y foco de entrada

Muchos de los controles que se introducen en un formulario, como pueden ser las cajas de texto, las listas y los botones, son capaces de tomar lo que se conoce como el *foco de entrada*. El componente que tiene el foco suele aparecer resaltado con un borde más grueso, caso de los botones; mostrando un cursor en su interior, como en las cajas de texto, o usando un recurso similar. Con ello indican que cualquier introducción de datos que efectuemos, usando el teclado, irá dirigida a ese control, al control que tiene el foco.

Cuando se abre un formulario el foco de entrada lo toma el primer control según el orden de acceso previamente establecido durante el diseño. Para pasar el foco de entrada de un control al otro, según ese orden, se usa el tabulador. En caso de que así interese, un control puede quedar fuera de ese acceso mediante el tabulador. Las propiedades implicadas son `TabIndex` y `TabStop`. La primera mantiene un índice que establece el orden de los controles, mientras que la segunda indica si un componente debe tomar o no el foco de entrada mediante el tabulador. En principio el orden de acceso a los controles viene determinado por el orden en el que se han insertado en el contenedor. No obstante, podemos modificar dicho orden alterando la citada propiedad `TabIndex`, aunque resulta mucho más cómodo usar la opción **Ver>Orden de tabulación**. Al seleccionarla, y suponiendo que tiene insertados varios controles en el formulario, verá aparecer junto a cada uno de ellos un recuadro con un número que, en realidad, es el contenido de la propiedad `TabIndex` (véase figura 10.14). El cursor cambia de forma al situarlo sobre los controles. Pulsando el botón principal del ratón iremos estableciendo el nuevo orden que deseemos.

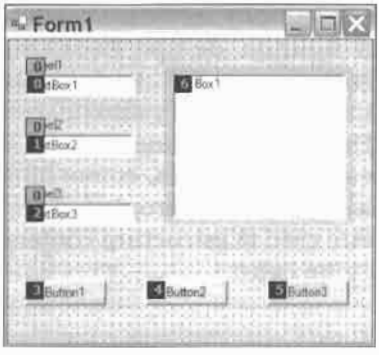


Figura 10.14. Establecemos el orden de acceso a los controles mediante el tabulador

Nota

Para volver al modo de diseño habitual, pudiendo insertar y personalizar los componentes, seleccione de nuevo la misma opción Ver>Orden de tabulación.

Si en el ejemplo de la figura 10.14 no deseamos que la lista, el control que hay a la derecha mayor que los demás, sea accesible mediante el tabulador no tenemos más que dar el valor `False` a la propiedad `TabStop` que, por defecto, siempre toma el valor `True`. Aunque demos el valor `False` a la propiedad `TabStop`, un control podrá seguir tomando el foco de entrada mediante otros medios, por ejemplo pulsando sobre él con el botón principal del ratón. Cualquier control que esté visible y activado, salvo algunas excepciones, puede tomar el foco de entrada. Para verificar si un control puede o no tomar el foco de entrada no hay más que comprobar el valor de su propiedad `CanFocus`. Si contiene `True` es que existe esa posibilidad, teniendo `False` en caso contrario.

También podemos determinar si un control tiene actualmente o no el foco de entrada consultando la propiedad `Focused`. Ésta, al igual que `CanFocus`, es de tipo `Boolean` y sólo de lectura. Si contiene el valor `True` significará que ese control tiene el foco de entrada.

Lo habitual es que el foco de entrada pase de un control a otro a demanda del usuario, ya sea usando el tabulador o la selección directa mediante el ratón o un acceso rápido. En ciertos casos, sin embargo, podemos necesitar establecer el foco de entrada desde código, como respuesta a alguna acción. Es algo que vimos en el primer ejemplo propuesto en este capítulo, en el que utilizábamos el método `Focus()` para desplazar el foco desde el botón, que lo obtenía al ser pulsado, a la caja de texto que había a la izquierda.

Anclaje y adosado de controles

Un aspecto que se ha mejorado considerablemente, comparando los servicios de formularios Windows con los formularios de Visual Basic 6, es el ajuste

automático de la posición y dimensiones de los componentes en el interior de sus contenedores. En Visual Basic 6 no era extraño tener que responder al evento `Resize` para, mediante cálculos matemáticos, ajustar los controles y conseguir una presentación consistente.

En Visual Studio .NET toda esa codificación manual puede evitarse gracias a la existencia de las propiedades `Anchor` y `Dock`.

La primera establece los puntos de anclaje del control respecto a su contenedor. Puede contener, mediante combinación lógica, uno o más de los valores enumerados en la tabla 10.2.

Por defecto todos los controles tienen los valores `Left` y `Top`, de tal manera que mantienen su posición estática en el formulario relativa a la esquina superior izquierda de éste.

Tabla 10.2. Valores de la enumeración `AnchorStyles`

Valor	Punto de anclaje
`None`	Ninguno
`Left`	Izquierdo
`Top`	Superior
`Right`	Derecho
`Bottom`	Inferior

Si desea que un cierto control conserve su posición relativa a la esquina inferior derecha, por ejemplo un botón que haya situado en esa posición, no tendría más que modificar la propiedad `Anchor` eliminando los valores `Left` y `Top` y asignando `Right` y `Bottom`. Puede hacer esto durante la fase de diseño en la ventana de propiedades (véase figura 10.15) o bien con una sentencia como la siguiente:

```
Button1.Anchor = AnchorStyles.Right Or AnchorStyles.Bottom
```

En caso de que la propiedad `Anchor` de un control tenga dos valores opuestos, como `Left` y `Right` o `Top` y `Bottom`, lo que conseguiremos es que el ancho o alto del control, respectivamente, se adecuen a los cambios de tamaño del contenedor. Puede hacer una prueba simple pero que le permitirá comprender fácilmente el funcionamiento de `Anchor`. Partiendo de un formulario como el de la figura 10.14, en el que existen varias cajas de texto y etiquetas, una lista y tres botones, haremos los cambios siguientes:

- Seleccionamos la lista y, desplegando el editor de la propiedad `Anchor` en la ventana **Propiedades**, activamos los valores `Right` y `Bottom` sin desactivar los dos que aparecen marcados por defecto.
- Elegimos conjuntamente los tres botones que hay en la parte inferior, desplegamos el mismo editor de la propiedad `Anchor` y desactivamos los valores `Left` y `Top` para activar `Right` y `Bottom`.

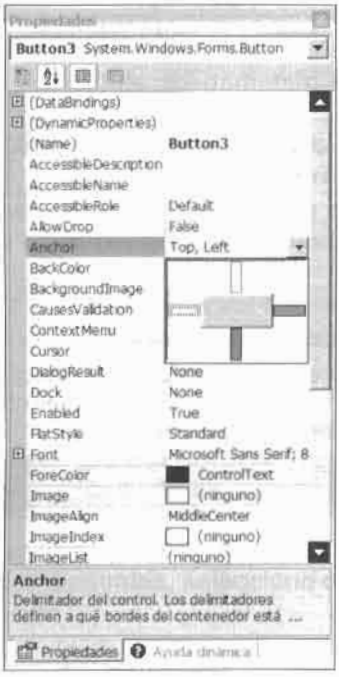


Figura 10.15. La propiedad `Anchor` dispone de un editor visual que facilita la modificación

Nota

Para seleccionar varios controles de manera conjunta puede pinchar sobre ellos con el botón principal del ratón mientras mantiene pulsada la tecla **Mayús**, o bien trazar con el botón principal del ratón un recuadro en el que estén incluidos esos controles.

Hechos esos cambios, al ejecutar el programa y alterar el tamaño de la ventana verá que las dimensiones de la lista se ajustan en consecuencia, ocupando siempre la mayor parte del espacio disponible. Los botones, por su parte, se desplazarán abajo y a la derecha para evitar la superposición sobre la lista. La figura 10.16 muestra la ventana, en ejecución, tras haber aumentado el ancho y alto.

Para conseguir esto mismo en Visual Basic 6 no teníamos más remedio que escribir código con el fin de realizar manualmente los ajustes.

Otra forma de mantener un control en una posición fija, independientemente de que las dimensiones del contenedor se vean alteradas en ejecución, consiste en adosarlo a un cierto margen de éste. Con ese fin se usa la propiedad `Dock` que, por regla general, toma por defecto el valor `None` de los enumerados en la tabla 10.3, aunque algunos controles, como puede ser `StatusBar`, asignan inicialmente un valor distinto.

Figura 10.16. Los controles de la ventana se ajustan automáticamente a los cambios de tamaño

Tabla 10.3. Valores de la enumeración `DockStyle`

Valor	El control se adosa
`None`	No se adosa
`Bottom`	A la parte inferior del contenedor
`Top`	A la parte superior del contenedor
`Left`	A la parte izquierda del contenedor
`Right`	A la parte derecha del contenedor
`Fill`	Ocupando todo el espacio disponible en el contenedor

Al igual que `Anchor`, la propiedad `Dock` dispone de un editor visual específico que permite seleccionar fácilmente el punto al que deseamos adosar el control. Inserte en el formulario un componente `StatusBar`, cuyo fin es servir como línea de estado en una ventana. Verá que se ajusta automáticamente a la parte inferior, sin importar los cambios en las dimensiones de la ventana. Puede modificar la propiedad `Dock` para situar la barra en la parte superior, izquierda, derecha u ocupar toda la ventana.

Usadas conjuntamente, estas dos propiedades simplifican enormemente el diseño de interfaces de usuario que se adaptan sin problemas a la modificación de las dimensiones iniciales de la ventana sin, por ello, tener que escribir código alguno.

Eventos de teclado y ratón

La mayoría de los controles que usaremos para diseñar interfaces de usuario Windows se encargan de gestionar adecuadamente los eventos generados

por teclado y ratón, principales dispositivos de introducción de información por parte del usuario. Nada nos impide, sin embargo, responder personalmente a esos eventos con código a medida.

Las actuaciones sobre el teclado son susceptibles de generar tres eventos diferentes: `KeyDown`, `KeyUp` y `KeyPress`. Los dos primeros se generan para todas las teclas en el momento en que se pulsan y se liberan, respectivamente. El último se produce cuando la tecla pulsada tiene asociado un código de carácter, es decir, no se trata de una tecla de desplazamiento del cursor o una tecla de función o edición. Los eventos `KeyUp` y `KeyPress` reciben como parámetro una estructura `KeyEventArgs`, en la que existen diversos miembros informativos. El más importante es `KeyCode`, conteniendo el código de la tecla pulsada o liberada. Además podemos comprobar si estaban pulsadas las teclas **Alt**, **Control** o **Mayús** mediante los miembros `Alt`, `Control` y `Shift`, respectivamente. Éstos contendrán el valor `True` en caso de que estuvieran pulsadas o bien `False` en caso contrario. El evento `KeyPress` recibe un parámetro de tipo `KeyPressEventArgs`, cuyo miembro `KeyChar` aloja el carácter correspondiente a la tecla pulsada.

En cuanto al ratón, los eventos más habituales son `MouseDown`, `MouseMove` y `MouseUp`. Se producen al pulsarse un botón del ratón sobre el control, desplazar el puntero del ratón o liberar el botón, respectivamente. Adicionalmente también se generan los eventos `MouseEnter`, `MouseHover` y `MouseLeave`, que denotan la entrada del puntero del ratón sobre el área que ocupa el control, la permanencia sobre él y la salida cuando el puntero se desplaza a otro punto.

Los tres primeros eventos reciben un parámetro con los miembros necesarios para saber qué botón ha sido el pulsado, cuál es la posición actual del puntero o bien la medida en que se ha desplazado la rueda del ratón si es que éste cuenta con una.

Nota

Usando el evento `MouseWheel` podemos actuar ante un desplazamiento de la rueda del ratón, sin necesidad de atender al resto de los eventos de este dispositivo.

La clase `Control` dispone de tres propiedades compartidas, que podemos utilizar a través de cualquier control o bien directamente disponiendo el nombre de la clase y un punto, que alojan información sobre el estado de las teclas **Alt**, **Control** y **Mayús**, el estado de los botones del ratón y la posición del puntero. Esas propiedades son `ModifierKeys`, `MouseButtons` y `MousePosition`. Las tres son sólo de lectura y podemos utilizarlas desde cualquier punto para actuar en consecuencia o modificar el comportamiento por defecto del control, sin necesidad de estar en el método asociado a uno de los eventos de ratón o teclado.

Con el fin de ver en la práctica cómo utilizar algunas de las propiedades y eventos mencionados, principalmente los de teclado y ratón, vamos a servir-

nos de un sencillo ejemplo. Partiremos de una nueva aplicación basada en formulario Windows, formulario en el que insertaremos dos controles `Button`, un `RadioButton` y un componente `Timer`. Este último aparecerá en la parte inferior del diseñador al tratarse de un elemento no visual.

Modificamos la propiedad `Text` de los primeros tres controles dejándola vacía. Ajustamos el tamaño y la posición de los botones para conseguir la apariencia de la figura 10.17 que, aunque vagamente, recuerda al clásico juego de la pelota y las dos paletas que es conocido como *Pong*. Modificamos la propiedad `Anchor` del botón situado a la derecha desactivando el valor `Left` y activando `Right`. De esta forma si se modifica el ancho del formulario en ejecución la paleta siempre se mantendrá en el margen derecho. También daremos el valor `True` a la propiedad `Checked` del `RadioButton`, a fin de que aparezca como seleccionado.

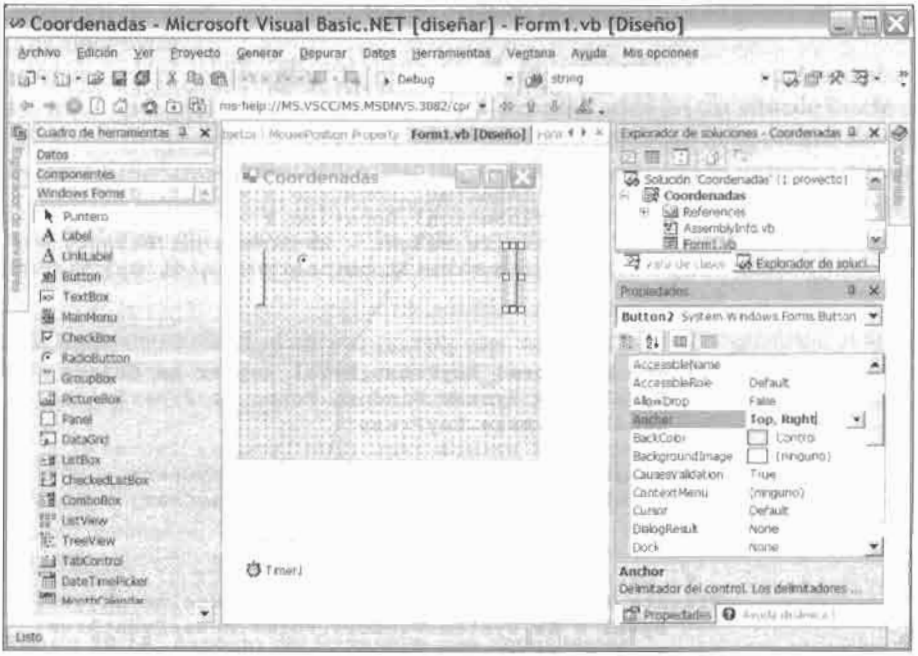


Figura 10.17. Aspecto del formulario en la fase de diseño

El componente `Timer` se usa para generar un evento periódico que, en este caso concreto, nos servirá para ir actualizando la posición del `RadioButton` simulando su movimiento por la ventana. Asignamos el valor `20` a la propiedad `Interval`, de tal forma que el evento se produzca cada 20 milisegundos. El valor inicial de la propiedad `Enabled`, que determina si el control genera o no el evento, inicialmente será `False`.

Por último, en lo que a establecimiento de propiedades se refiere, daremos el valor `True` a la propiedad `KeyPreview` del propio formulario. De esta manera solicitamos que cuando se pulse una tecla, que debería ir dirigida al control que tenga el foco de entrada en ese momento, los eventos correspondientes

se generen primero en el formulario, lo cual nos permitirá controlarlos de manera general sin tener que gestionar el evento `KeyPress` de los dos botones y el `RadioButton`.

A continuación asociaremos el código siguiente a los eventos `Tick` del control `Timer` y `KeyPress`, `MouseMove` y `MouseUp` del formulario.

```
' Estas variables permiten determinar el sentido
' en que se mueve el RadioButton. Si dx = 1 se mueve
' hacia la derecha
Private dx As Integer = 1, dy As Integer = 1

' dx y dy dan el paso desde el actual
Private Sub Timer1_Tick(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Timer1.Tick
  ' Actualizamos la posición del RadioButton
  RadioButton1.Left += dx
  RadioButton1.Top += dy

  ' Dependiendo de la posición del RadioButton actualizamos
  ' las variables que controlan el movimiento para
  ' el rebote en los bordes del formulario.
  dx = IIf(RadioButton1.Location.X >= _
      Me.ClientSize.Width - RadioButton1.Width, -1, dx)
  dx = IIf(RadioButton1.Location.X = 0, 1, dx)
  dy = IIf(RadioButton1.Location.Y >= _
      Me.ClientSize.Height - RadioButton1.Height, -1, dy)
  dy = IIf(RadioButton1.Location.Y = 0, 1, dy)
End Sub

' Al detectar una pulsación de tecla
Private Sub Form1_KeyPress(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyPressEventArgs) _
    Handles MyBase.KeyPress
  ' Mueve la paleta izquierda cada vez que pulsa Q o Z
  Button1.Top -= IIf(Char.ToUpper(e.KeyChar) = "Q", 5, 0)
  Button1.Top += IIf(Char.ToUpper(e.KeyChar) = "Z", 5, 0)
End Sub

' Al detectarse movimiento del ratón en el formulario
Private Sub Form1_MouseMove(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.MouseEventArgs) _
    Handles MyBase.MouseMove
  ' movemos la paleta derecha
  Button2.Top = e.Y
End Sub

' Al detectarse la liberación de un botón del ratón
Private Sub Form1_MouseUp(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.MouseEventArgs) _
    Handles MyBase.MouseUp
  ' Si se ha pulsado el botón izquierdo, activamos el Timer
  Timer1.Enabled = e.Button = MouseButtons.Left
End Sub
```

Los comentarios introducidos describen la finalidad de cada método. Básicamente, activamos y desactivamos la generación del evento `Tick` por parte

del `Timer` mediante la pulsación de los botones del ratón, mientras que el desplazamiento de éste y la pulsación de ciertas teclas modifican la posición vertical de los botones. El `RadioButton` va desplazándose gracias al código asociado al evento `Tick`, código que habría que completar para detectar la colisión con los botones y evitar que la supuesta pelota los atraviese.

Arreglos de controles

Una técnica bastante habitual en Visual Basic 6, a la hora de crear interfaces de usuario complejas con multitud de controles, consiste en crear arreglos de controles. No hay más que copiar un control y después pegarlo para crear ese arreglo con varios controles del mismo tipo y con el mismo nombre, diferenciándose unos de otros por un índice almacenado en la propiedad `Index`. Esta posibilidad ya no existe en Visual Basic .NET o, para ser más exactos, digamos que no es necesaria.

La construcción de arreglos de controles simplificaba determinados trabajos al permitir, por ejemplo, la edición de sus propiedades en un simple bucle o la codificación de gestores de eventos genéricos, evitando así la apertura de un método para cada uno de los controles. Para verlo en la práctica vamos a partir de una interfaz simple pero relativamente tediosa: en un formulario necesitamos ocho botones que, teóricamente, efectuarán una operación distinta sobre un mismo objeto. Esta interfaz podría ser la de la figura 10.18.

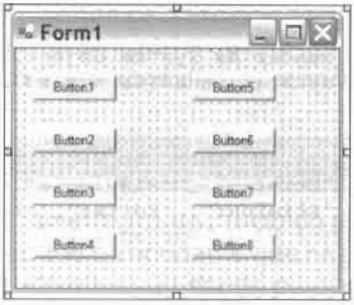


Figura 10.18. Aspecto del formulario que necesitamos

Insertar ocho botones iguales no es un trabajo que requiera mucho esfuerzo, ciertamente, pero ahora es necesario establecer un título para cada uno de ellos, ajustar propiedades de anclaje, color y, en general, adaptar su apariencia. Lo mismo ocurrirá con el comportamiento, debiendo escribir un método asociado al evento `Click` de cada botón para, en definitiva, llamar a un método diferente de un mismo objeto según el supuesto que se ha propuesto hace un momento.

Si ese trabajo de personalización e implementación de funcionalidad lo extrapolamos de este ejemplo, bastante sencillo, a una aplicación real, con cientos de componentes, el esfuerzo lógicamente no será tan nimio. Para reducirlo podemos utilizar algunas de las técnicas que van a describirse en los puntos siguientes.

La clase `Form`, al igual que otras derivadas de `Control`, dispone de una propiedad de tipo `ControlCollection`. En el caso de `Form` esa propiedad se llama `Controls` y, como puede suponer, contiene la colección de controles que hay insertados en el formulario.

Anteriormente vio que el diseñador, a medida que insertamos controles en el formulario, se ocupa de generar el código necesario para crear los controles y añadirlos a esa colección.

Al igual que cualquier otra colección, los elementos de `ControlCollection` pueden ser enumerados mediante `For Each/Next`, sabiendo que cada elemento será un objeto de la clase `Control` o de una clase derivada. También podemos saber el número de elementos de la colección consultando la propiedad `Count` y usar la propiedad `Item` como si de un arreglo se tratase, accediendo a los elementos de manera individual.

No hay que olvidar que la colección mantiene los elementos bajo el tipo genérico `Control`, lo cual hace posible el uso polimórfico de todos ellos. En ocasiones, sin embargo, puede ser necesario comprobar el tipo real, por ejemplo mediante el operador `TypeOf`, o incluso convertir la referencia genérica al tipo adecuado, mediante la función `CType()`.

Suponga que, tomando como punto de partida el formulario de la figura 10.18, desea establecer un título individual para cada botón. Haga doble clic sobre el fondo del formulario para abrir el método asociado al evento `Load` y, a continuación, introduzca el código siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
          ByVal e As System.EventArgs) Handles MyBase.Load

  Dim UnControl As Control

  Dim Titulo() = {"Primero", "Anterior", "Siguiente", "Último", _
                  "Añadir", "Eliminar", "Editar", "Aceptar"}
  Dim Indice As Integer


  For Each UnControl In Me.Controls

    If TypeOf UnControl Is Button Then

      UnControl.Text = Titulo(Indice)
      Indice += 1
    End If
  Next
End Sub
```

Usando esta misma técnica podríamos establecer las propiedades que fueran necesarias para cualquier conjunto de controles. La colección `ControlCollection` no tiene las limitaciones existentes en los arreglos de controles de versiones previas de Visual Basic, ya que los controles no tienen por qué tener el mismo nombre, ni siquiera tienen que ser del mismo tipo.

Uso compartido de gestores de eventos

Casi más interesante que el acceso genérico a las propiedades o métodos de los controles, utilizando el sistema que acaba de explicarse, resulta la posibilidad de utilizar un único método como gestor de eventos de varios controles o, incluso, varios eventos distintos siempre que el tipo de delegado sea compatible. Esto puede ahorrarnos bastante trabajo al evitar la codificación repetitiva de una lógica similar o, incluso, idéntica.

Si estamos asignando los gestores de eventos desde el diseñador de visual, lo que haremos será crear el gestor para el primer evento del primer control usando el método habitual: doble clic sobre el control si es para usar el evento por defecto o despliegue de la lista de eventos que hay en la parte superior del editor. Después bastará con ir añadiendo tras la cláusula `Handles` el nombre de los controles y eventos a gestionar.

Haga doble clic sobre el primero de los botones del formulario del ejemplo anterior y, a continuación, modifique el código para que quede así:

```
Private Sub Button1_Click( _
  ByVal sender As System.Object, ByVal e As System.EventArgs) _
  Handles Button1.Click, Button2.Click, Button3.Enter
  MessageBox.Show(CType(sender, Button).Text)
End Sub
```

Observe que hemos añadido el evento `Click` del segundo botón y el evento `Enter` del tercero a la lista de eventos a controlar por este gestor. En su interior nos limitamos a mostrar en una pequeña ventana el título del botón sobre el que se ha pulsado o que ha generado el evento `Enter`. Al ejecutar el programa podrá ver cómo el mismo método gestiona esos tres eventos.

Nota

El parámetro `sender` recibido por todos los gestores de eventos es una referencia al componente que ha generado el evento, el dato que nos permitirá determinar de dónde procede ese evento y actuar en consecuencia.

Como ya sabe, el uso de la cláusula `Handles` tiene como alternativa la sentencia `AddHandler` que conocimos en un capítulo previo. Tomando como base el código del punto anterior, en el que utilizamos un bucle para establecer los títulos de los botones, añada la siguiente línea al interior del bucle:

```
AddHandler UnControl.Click, AddressOf ClicBoton
```

El método `ClicBoton`, ahora mismo inexistente, sería el encargado de gestionar el evento `Click` de los ocho botones que hay en el formulario, diferenciando uno de otro mediante el parámetro `sender`:

```
Private Sub ClicBoton(ByVal sender As Object, _
                      ByVal e As System.EventArgs)
```

```
    Select Case CType(sender, Button).Text
      Case "Anterior"
        MessageBox.Show("Nos vamos a la fila anterior")
      Case "Siguiente"
        MessageBox.Show("Nos vamos a la fila siguiente")
      Case "Primero"
        MessageBox.Show("Nos vamos a la primera fila")
      Case "Ultimo"
        MessageBox.Show("Nos vamos a la última fila")
      Case "Añadir"
        MessageBox.Show("Añadimos una nueva fila")
      Case "Eliminar"
        MessageBox.Show("Eliminamos la fila actual")
      Case "Editar"
        MessageBox.Show("Editamos la fila actual")
      Case "Aceptar"
        MessageBox.Show("Aceptamos los cambios")
    End Select
  End Sub
```

En la práctica, como es lógico, no nos limitaríamos a mostrar un mensaje sino que llevaríamos a cabo la tarea que correspondiese al botón pulsado. Lo importante es que la técnica para compartir el gestor e identificar la procedencia del evento es la misma que acaba de describirse.

Creación de controles en ejecución

Ya que podemos establecer propiedades y asignar gestores a eventos de manera genérica, usando la colección `ControlCollection` y las técnicas aprendidas en los puntos anteriores, la única tarea que tendríamos que efectuar desde el diseñador, a la hora de componer una interfaz de usuario, es la inserción de los componentes. Habitualmente resulta más cómodo hacer este trabajo mediante operaciones de arrastrar y soltar que escribiendo código, pero eso no significa que no podamos hacerlo.

Los componentes son objetos y, como tales, podemos crearlos a partir de sus respectivas clases, como hemos hecho repetidamente en capítulos previos con clases que no eran componentes. Una vez creados, no tenemos más que utilizar el método `Add()` de la colección `ControlCollection` para añadirlos al formulario o, en su caso, al contenedor que corresponda.

Para ver un ejemplo puede partir de un nuevo formulario en blanco, hacer doble clic sobre él e insertar el código siguiente que, como puede ver, recrea los mismos ocho botones que teníamos en el ejemplo usado hasta ahora. La diferencia es que todo el trabajo se efectúa mediante código.

```
For Indice = 1 To 8

  UnControl = New Button()
  With UnControl

    .Text = Titulo(Indice - 1)

    .Top = ((Indice - 1) Mod 4) * (.Height + 20) + 20
```

```
        .Left = IIf(Indice <= 4, 20, 40 + UnControl.Width)
    End With

    Controls.Add(UnControl)
  Next
```

La posición la calculamos a partir del índice del bucle y utilizando operadores aritméticos y lógicos para disponerlos en dos columnas de cuatro filas.

Herencia visual

Cada vez que añadimos un nuevo formulario a nuestro proyecto estamos derivando una clase tomando como base la clase `Form` que, como ya sabe, es un formulario genérico, vacío. Cuando va a desarrollarse un proyecto importante es habitual usar un diseño base común para todas las ventanas, por ejemplo contando con un fondo tipo anagrama de empresa, unos botones comunes de ayuda o cierre, etc. Estos elementos, con lo que hemos visto hasta ahora, tendríamos que añadirlos en cada nuevo formulario, representando un trabajo repetitivo.

Visual Basic .NET cuenta con una posibilidad, inexistente en versiones previas, que podríamos denominar como *herencia visual*. Se trata de crear un formulario base y, posteriormente, utilizarlo para derivar a partir de él el resto de los formularios del proyecto. El formulario base puede alojarse en el mismo proyecto, que seguramente es lo ideal, o bien disponerse en un ensamblado independiente, caso en el cual sería necesario importar el ámbito correspondiente.

Es fácil de entender la utilidad y el uso de la herencia visual simplemente poniéndola en práctica. Iniciamos un nuevo proyecto de aplicación para Windows y, en el formulario que aparece, insertamos unos botones en la parte inferior derecha, modificando sus títulos y la propiedad `Anchor` para que permanezcan siempre ajustados en esa posición. Si lo deseamos, podemos asociar algo de código al evento `Click` de cada uno de ellos. Antes de continuar hay que compilar la aplicación.

Acto seguido desplegamos el menú **Proyecto** y luego seleccionamos la opción **Agregar formulario heredado**. En la primera ventana que aparece introducimos el nombre que se dará al nuevo formulario, tras lo cual veremos el cuadro de diálogo de la figura 10.19. En él aparecen todos los formularios existentes en el proyecto actual y que podemos utilizar como base del nuevo. También podríamos pulsar el botón **Examinar** para abrir cualquier otro módulo que contenga un formulario para derivar de él.

El nuevo formulario mostrará exactamente la misma apariencia que el tomado como base. Además, los controles insertados en éste aparecen en el nuevo bloqueados, de forma que no podemos alterar sus propiedades. En ese momento podemos personalizar el formulario, por ejemplo añadiendo los elementos específicos para la funcionalidad que vaya a tener. Parte del trabajo, sin embargo, ya lo hemos heredado del formulario base. Si comprueba el código de éste, como se ha hecho en la figura 10.20, verá que la clase base no es `System.Windows.Forms.Form` sino la clase de formulario creada con anterioridad.

Figura 10.19. El cuadro de diálogo para seleccionar la base de un formulario

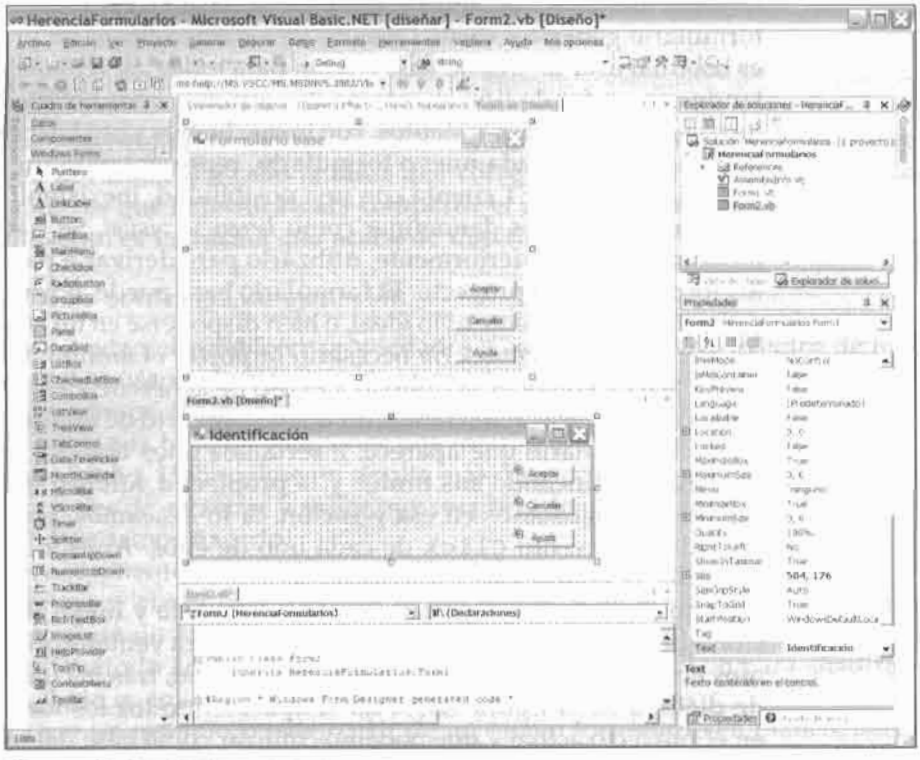


Figura 10.20. El formulario base en la parte superior, el derivado en el centro y su código en la parte inferior

Puntos clave

- Los formularios Windows son parte de los servicios que la plataforma Microsoft .NET ofrece para el sistema operativo Windows, servicios que pueden ser usados desde todos los lenguajes .NET.

- La clase base de todos los formularios es `Form`, definida en el ámbito `System.Windows.Forms`. Dicha clase implementa la funcionalidad básica de una ventana Windows.
- Visual Basic .NET cuenta con un diseñador visual que facilita la creación de formularios mediante operaciones de arrastrar y soltar y la edición de propiedades.
- Las aplicaciones basadas en formularios Windows utilizan un objeto de la clase `Application` para controlar todo el proceso.
- Una aplicación Windows puede contar con múltiples ventanas, ya sean independientes, de tipo MDI o cuadros de diálogo.
- Al iniciar una aplicación Windows automáticamente aparecen en el Cuadro de herramientas los controles usados más habitualmente. Utilizando el menú emergente podemos añadir cualesquiera otros controles que pudiéramos precisar.
- Todos los controles Windows están derivados de la clase `Control`, por lo que cuentan con un conjunto común de miembros: propiedades, métodos y eventos.
- Las nuevas características de los formularios Windows, respecto a los formularios de Visual Basic 6, hace que el diseño de interfaces de usuario sea más fácil.
- Los controles pueden anclarse de manera relativa a su contenedor, adaptando su posición y dimensiones en caso de que el tamaño de dicho contenedor sea modificado.
- Todos los controles disponen de eventos comunes que permiten responder a la introducción de datos desde el teclado y el ratón.
- Los formularios disponen de una propiedad, llamada `Controls`, de tipo `ControlCollection` en la cual se almacena la colección de controles existentes en la ventana.
- Usando dicha propiedad es posible modificar los controles existentes, así como añadir otros nuevos.
- Mediante la herencia visual es posible crear nuevos formularios derivándolos no de `System.Windows.Forms.Form` sino de clases de formularios propias, ahorrando así parte del trabajo de diseño.

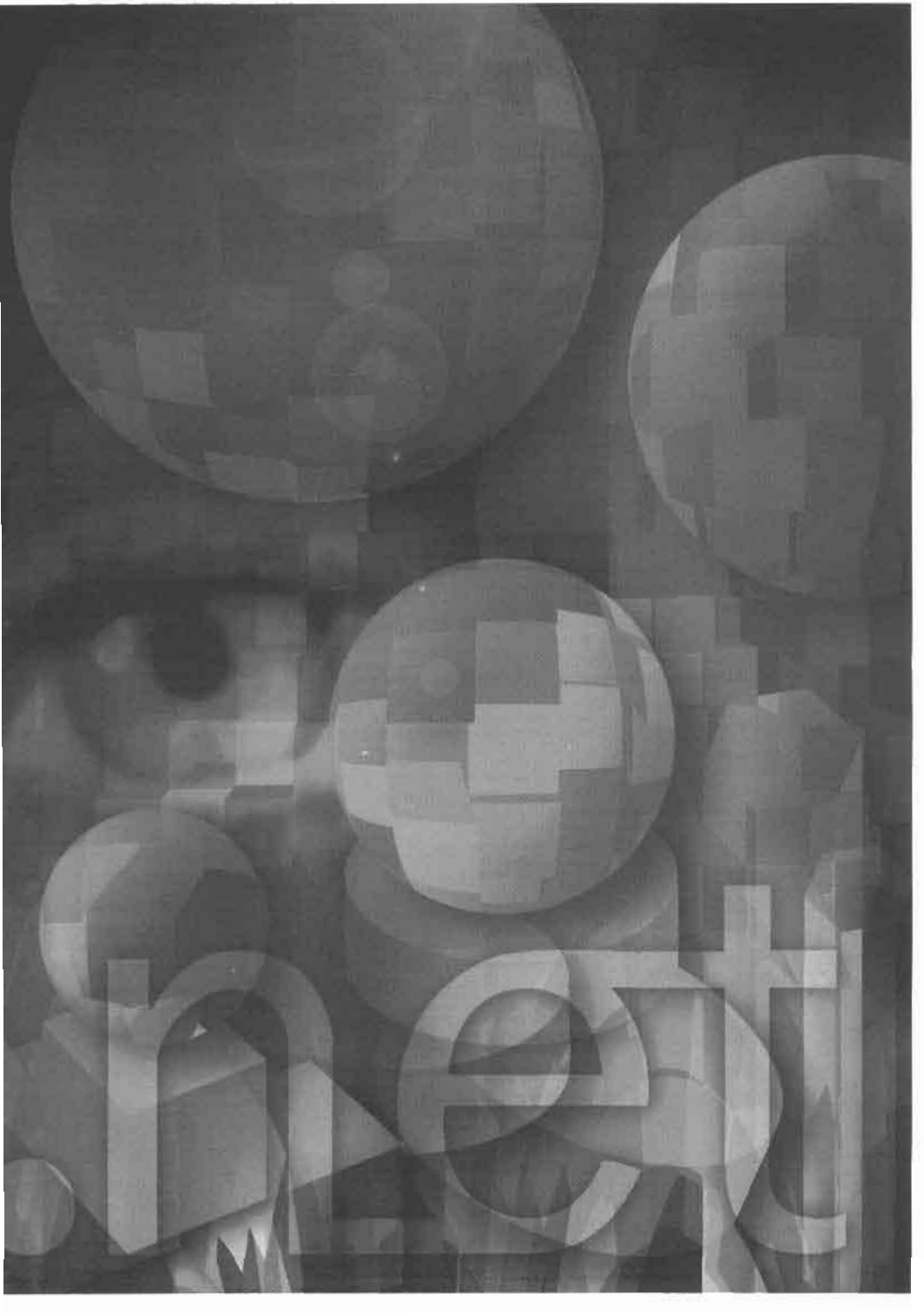
Resumen

Salvo las diferencias propias debidas a los cambios que se han producido en el lenguaje, y que causan, por ejemplo, que el formulario y los controles puedan ser tratados de manera más flexible gracias a las características de orientación a objetos, lo cierto es que el diseño de interfaces de usuario y, en general,

el desarrollo de aplicaciones con Visual Basic .NET se asemeja bastante al que conocíamos de versiones previas. Básicamente no hay más que arrastrar y soltar componentes, personalizarlos mediante la edición de propiedades y asociar la funcionalidad mediante los eventos.

Lógicamente, es necesario conocer todos los componentes con que cuenta Visual Basic .NET para poder aprovecharlos en nuestros proyectos. En este capítulo, sin embargo, nos hemos centrado más en el proceso general del trabajo, sin entrar a describir todos los detalles de cada objeto lo cual requeriría un espacio considerable. Esa información, además, ya la encontramos en forma de referencia en la propia ayuda del producto.

Tras leer este capítulo, por tanto, ha adquirido las nociones necesarias para comenzar a desarrollar aplicaciones Windows, sirviéndose para ello del apoyo que supone la documentación electrónica.


next

11

Formularios Web

Cada vez es mayor el número de empresas que optan por usar la arquitectura de Internet, sus protocolos y servicios, como medio para conectar los sistemas de su propia empresa, así como la propia Internet como infraestructura de comunicación para conseguir esa conexión. En un entorno como el de Internet no podemos asumir que todos los clientes tengan un PC, funcionando con Windows como sistema operativo, para poder ejecutar las interfaces de usuario basadas en formularios Windows descritas en el capítulo previo. Existen otros dispositivos y otros sistemas operativos, potenciales clientes a los que hay que ofrecer otro tipo de interfaz de acceso a nuestros servicios.

Una de las novedades más interesantes de Visual Studio .NET, realmente de la plataforma Microsoft .NET, es ASP.NET, el sucesor de ASP (*Active Server Pages*). Mediante ASP.NET podemos desarrollar aplicaciones, generalmente interfaces de usuario, accesibles desde cualquier cliente que sea capaz de acceder a un servidor Web. ASP.NET se encarga de ejecutar la lógica en el servidor y generar el código HTML apropiado para el cliente que efectúe la solicitud, sin importar el tipo de dispositivo, hardware o sistema operativo en que esté ejecutándose.

El objetivo de este capítulo es describir una parte de ASP.NET, concretamente los formularios Web. Con ellos podemos crear interfaces usando prácticamente la misma metodología descrita en el capítulo previo, arrastrando y soltando controles sobre un contenedor, editando propiedades y asociando código a eventos. En un capítulo posterior nos ocuparemos de otros elementos de ASP.NET pensados para la construcción de servicios Web.

Para poder seguir los ejemplos de este capítulo es imprescindible que tenga acceso a un equipo en el que esté instalado IIS, el servidor web de Microsoft. Éste se instala por defecto en las versiones *Server* de Windows NT 4.0 y Windows 2000, pero no en las versiones *Professional*, por lo que posiblemente tenga que acceder a la configuración del sistema y añadir este componente.

De ASP a ASP.NET

Desde su aparición, las páginas ASP supusieron un gran cambio en el desarrollo de sedes Web con contenido dinámico. Un documento web actual contiene tanto el diseño visual o estético de la página, generalmente en HTML, como código de guiones o *script*, habitualmente en JScript o VBScript. Este código puede ser ejecutado en el cliente o bien en el servidor, según los casos. El proceso en el servidor tiene la ventaja de que el cliente sólo recibe HTML simple, dejando la generación dinámica como una tarea del servidor Web.

Las páginas ASP se procesan en el servidor, habitualmente IIS (*Internet Information Server*). Éste ejecuta el código y combina su resultado con el contenido HTML original del documento, generando una página dinámicamente que es lo que se envía al cliente. Las ventajas de ASP son muchas facilitando, por ejemplo, la generación dinámica de contenido a partir de información almacenada en una base de datos sin necesidad de escribir un CGI (*Common Gateway Interface*) clásico. Basta con algo de código embebido en el documento para conseguir un resultado rápido.

Los inconvenientes de ASP, no obstante, también tienen su peso. En un documento ASP se combina el código HTML con contenido estático y marcas de estilo con código JScript, VBScript, o incluso ambos, que generan el contenido no estático. Diferentes lenguajes y sintaxis entremezclados en un mismo documento, algo difícil de mantener. Además, hay que tener en cuenta que el código de los documentos ASP es interpretado cada vez que se solicita la página.

Nota

ASP está estrechamente ligado a Windows, concretamente al servidor web de Microsoft, el conocido IIS. En otros sistemas, como es el caso de Linux, han aparecido unas soluciones similares que, como PHP, tienen sus bases en ASP.

Naturaleza de un documento ASP

Una página ASP típica es un documento, con extensión `ASP`, en el que se combinan el código HTML con guiones que el servidor web debe interpretar y ejecutar antes de enviar la respuesta al cliente. El siguiente fragmento corresponde a un documento ASP real, concretamente a parte del documento por defec-

to de *Torre de Babel* (`http://www.fcharte.com`), una sede que el autor mantiene y cuya portada puede ver en la figura 11.1.

```
<%@ language="VBScript" %>
<head>
 <meta http-equiv="Content-Type" content="text/html;
   charset=windows-1252">
<title>Torre de Babel
   <% If Request.QueryString = "" Then
        Response.Write("(Portada)")
      Else
        Response.Write("(" & Request.QueryString & ")")
      End If
   %>
 </title>
 <link rel="stylesheet" type="text/css" href="TorreBabel.css">
</head>
<body>
   ...
```

Como puede ver, al inicio del documento se indica el lenguaje en el que están codificados los guiones del documento, en este caso VBScript. A continuación se combinan las marcas HTML corrientes, como `<head>` o `<title>`, con código que debe ser interpretado por el servidor, como el condicional delimitado por las parejas de caracteres `<%` y `%>` que, en este caso concreto, introduce en el documento la cadena de caracteres `"(Portada)"` o bien el valor que contenga la propiedad `QueryString` del objeto `Request`.

La inclusión de código ejecutable en el mismo documento que alberga el contenido y diseño de la página da lugar a lo que se conoce como *spaguetti code*, una mezcla de diversos lenguajes que se entremezclan a lo largo de todo el documento y que no contribuyen en absoluto a su legibilidad y facilidad de mantenimiento.

Aunque podemos usar JavaScript en lugar de VBScript, como una alternativa de preferencias respecto al lenguaje, no existen muchas más posibilidades a la hora de crear documentos ASP y el proceso siempre será el mismo: el documento se procesa en el servidor, éste toma el código de los guiones, lo interpreta y ejecuta y lo sustituye en el documento original por el resultado que se haya obtenido de esa ejecución.

Ocultación del código en ASP.NET

Uno de los cambios de ASP a ASP.NET es lo que Microsoft denomina *code-behind*, consistente en la ocultación del código ejecutable separándolo del contenido estático y diseño de la página. Para ello, en la cabecera del documento ASP.NET, se indicará cuál es el lenguaje en el que está escrito el código y facilitará el camino al archivo que contiene dicho código.

La sintaxis sería la siguiente:

```
<%@ Page Language="vb" Codebehind="WebForm1.aspx.vb"
    Inherits="WebSimple.WebForm1"%>
```

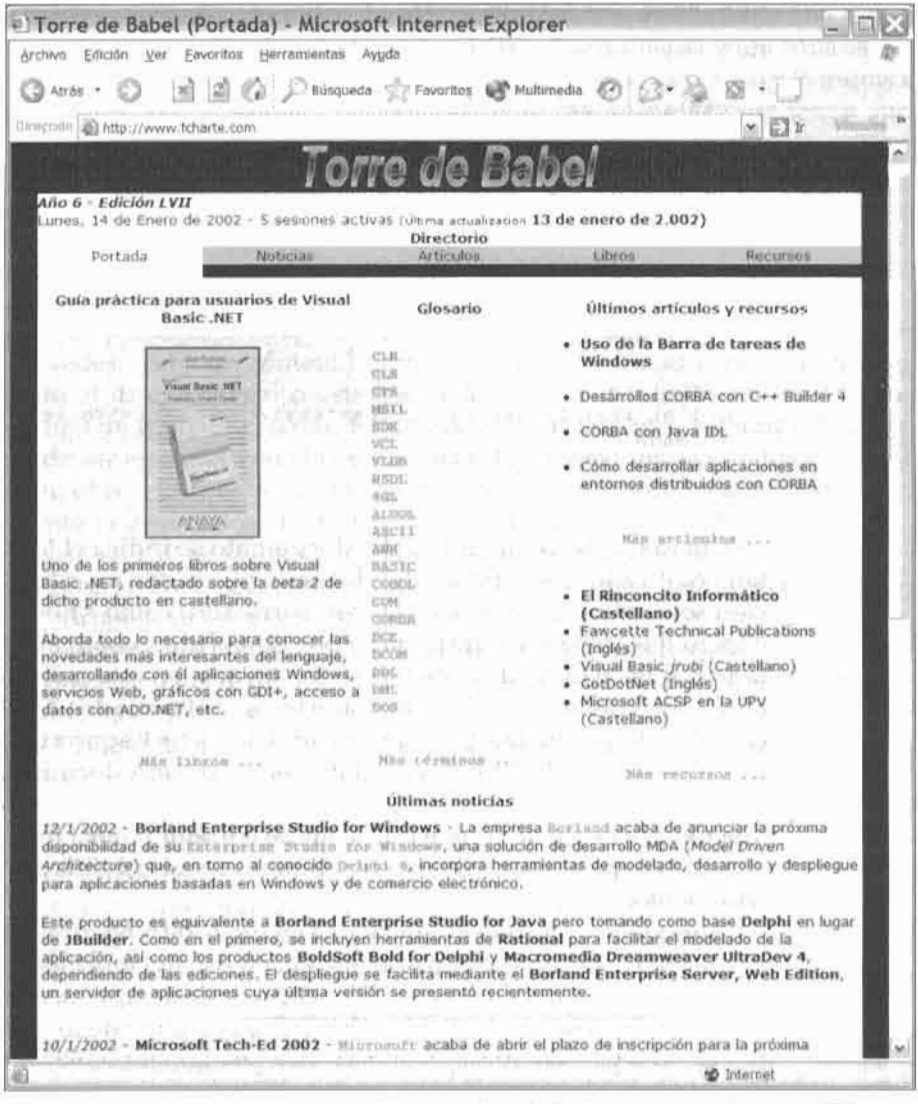


Figura 11.1. Aspecto de una web generada dinámicamente utilizando ASP

En este caso se indica que el código utilizado para implementar la lógica es Visual Basic .NET y está alojado en el archivo `WebForm1.aspx.vb`. Observe también que la habitual marca `<%` ahora es `<%@`, en este caso seguida de la definición de un objeto `Page` derivado de la clase `WebSimple.WebForm1`. En un momento veremos de dónde sale esa clase y qué es un objeto `Page`.

La extensión usada para los documentos ASP.NET no es `asp` sino `aspx`, estableciendo así una diferenciación clara que evite conflictos. En este caso, por tanto, el diseño y contenido estático de la página se encontraría en el archivo `WebForm1.aspx`, mientras que el código, en forma de objetos y métodos, se alojaría en `WebForm1.aspx.vb`.

Que exista la posibilidad de separar lógica y diseño en dos archivos separados no se convierte en una exigencia, es decir, en un documento ASP.NET, en el archivo con extensión aspx, es posible seguir introduciendo el código si es que así lo deseamos, aunque no será lo habitual salvo casos muy simples como algunos de los ejemplos que se propondrán a continuación.

Libertad de elección de lenguaje

Asumiendo que para el desarrollo de nuestros componentes y aplicaciones utilizásemos habitualmente Visual Basic .NET, por poner un ejemplo, ¿por qué tenemos que usar obligatoriamente JScript o VBScript a la hora de crear un documento ASP? Con ASP.NET esta obligación, o limitación, encuentra su fin. Ahora podemos usar como lenguaje de *script* prácticamente cualquiera de los lenguajes .NET si exceptuamos Visual C++. Esto significa que podemos utilizar C# o Visual Basic .NET, además de JScript .NET, para generar el contenido dinámico.

Nota

En ASP.NET el lenguaje VBScript ha desaparecido y, en su lugar, utilizaríamos Visual Basic.NET, que es un superconjunto de VBScript.

Las aplicaciones ASP.NET están constituidas por módulos, archivos con HTML y código de *script*, con extensión aspx. Cada uno de esos módulos se inicia, como se ha indicado en el punto anterior, con un encabezado en el que se indica el lenguaje que se utilizará. Este puede ser cualquiera de los disponibles en Visual Studio .NET a excepción de Visual C++. Incluso es posible utilizar lenguajes añadidos posteriormente al entorno, como COBOL, Pascal, Perl, Python o Java.

Elegir un lenguaje u otro, a la hora de escribir la lógica de un documento ASP.NET es, por tanto, una cuestión de gustos, una opción personal y no una imposición del fabricante, lo cual supone un cambio sustancial respecto a ASP, PHP y las soluciones que se utilizan hasta ahora.

Adiós a los guiones interpretados

Según se apuntaba anteriormente, el código embebido en los documentos ASP, en forma de guiones, es interpretado cada vez que un cliente solicita esa página. Esto compromete el rendimiento del servidor que, a pesar de ser bueno, obviamente no es el mejor posible debido a que se emplea parte de su tiempo en el análisis e interpretación del código, siempre como paso previo a la ejecución y generación final de la página a remitir a dicho cliente.

Con ASP.NET el escenario es distinto, ya que el código embebido en ellos es compilado la primera vez que es solicitada la página, tras lo cual el correspondiente ensamblado se almacena temporalmente en un espacio conocido como

cache. De esta forma, ante nuevas solicitudes de clientes el código puede ser ejecutado sin necesidad de recompilación.

Para comprobar esta afirmación puede efectuar una prueba muy sencilla. Abra el Bloc de notas, introduzca el código mostrado a continuación y guárdelo en un archivo, llamándolo `HolaMundo.aspx`, en la carpeta principal de su servidor Web, que habitualmente es `InetPut\wwwroot`.

```
<%@ Page Language="VB" %>
<html>
<body>
 <% Dim I As Byte
    For I=1 To 5
      Response.Write("<h" & I.ToString() & _
                     ">Hola mundo")
    Next
 %>
</body>
</html>
```

Básicamente es un documento ASP en el que se indica, al inicio, el lenguaje que va a usarse para introducir código. La primera vez que solicite esta página, tal y como se ha hecho en la figura 11.2, comprobará que la respuesta tarda unos segundos. Se está compilando el código obteniendo una versión ejecutable. Posteriormente, cualquier solicitud obtendrá una respuesta inmediata.

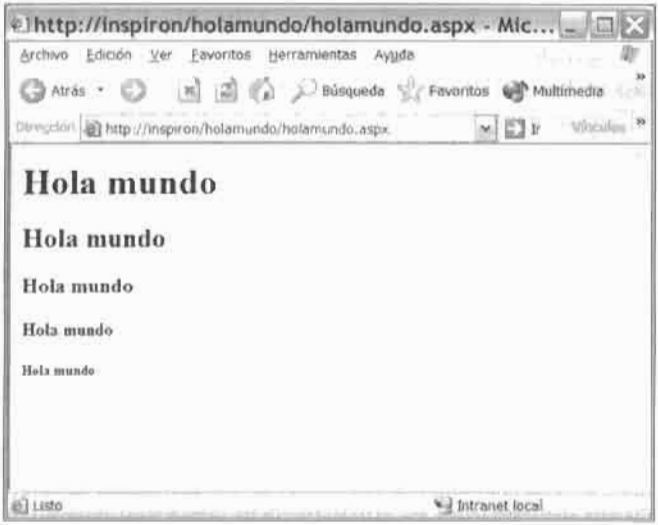


Figura 11.2. Página obtenida al solicitar el documento `HolaMundo.aspx`

En este caso hemos embebido el código del guión en el propio documento ASP.NET, de tal forma que la compilación la dejamos en mano del servidor que la efectuará a demanda, cuando lo necesite. Estamos usando la compilación dinámica de ASP.NET. La alternativa es separar el guión en un módulo separado que, compilado en el momento de distribuirse, sería accesible desde

el documento ASP.NET. De esta forma la compilación sería estática y el código ejecutable siempre, sin pérdida de tiempo.

Inicie un nuevo proyecto de tipo Aplicación Web ASP.NET, como se hace en la figura 11.3, usando el lenguaje Visual Basic.NET. Al pulsar el botón **Aceptar** se creará en el directorio raíz de IIS una carpeta para la nueva aplicación. Acto seguido nos encontraremos con un entorno como el de la figura 11.4. Es un formulario Web, en el cual podemos insertar componentes y establecer propiedades, como hicimos en el capítulo previo para crear aplicaciones Windows.

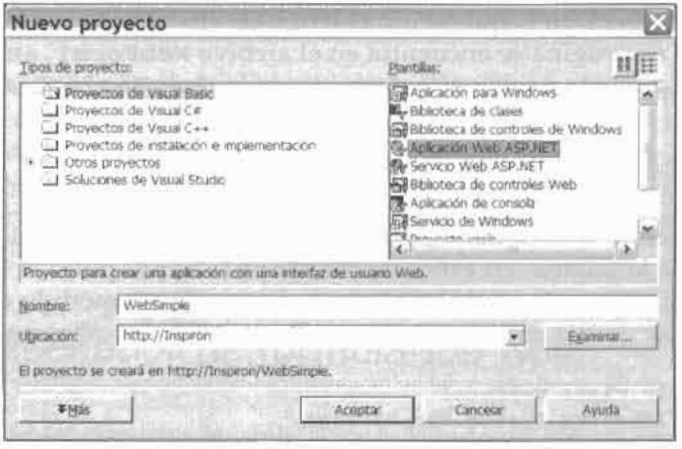


Figura 11.3. Iniciamos una aplicación Web ASP.NET

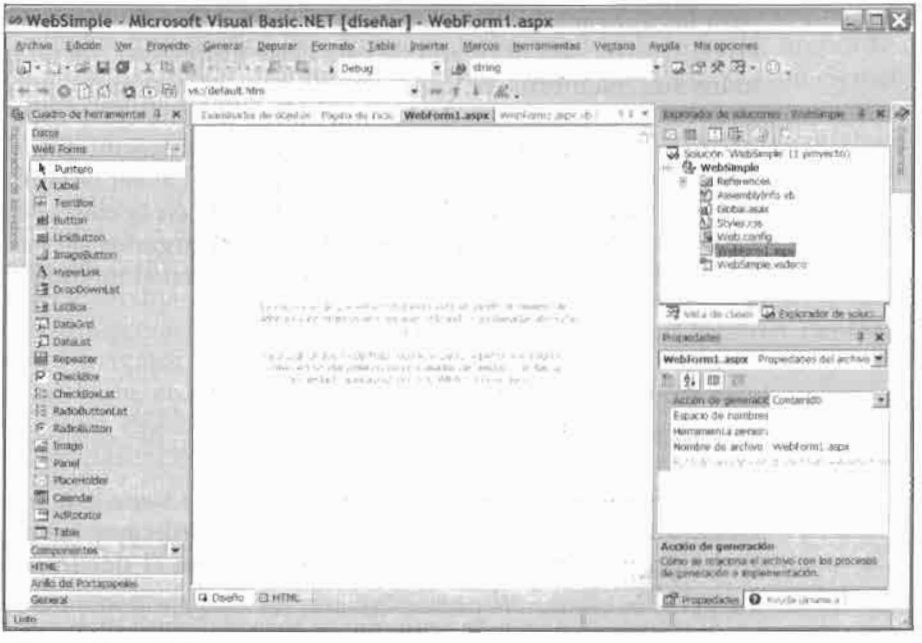


Figura 11.4. Entorno de diseño de un formulario Web

Pulse el botón **HTML** que hay en la parte inferior del editor, para acceder al código HTML del documento. Aparte de algunas etiquetas informativas y un formulario, lo más interesante del documento se encuentra en su cabecera:

```
<%@ Page language="vb"
    Codebehind="WebForm1.aspx.vb"
    AutoEventWireup="false"
    Inherits="WebSimple.WebForm1" %>
```

Este encabezado identifica el código que hay en el documento como una página ASP.NET en la que se usa el lenguaje Visual Basic. Se comunica que la lógica de esa página se encuentra en el archivo `WebForm1.aspx.vb` y, además, se indica que la página está derivada de la clase `PaginaSimple.WebForm1`.

Enlace entre el documento y el código

Teniendo seleccionado en el Explorador de soluciones el elemento correspondiente a la página, en este caso `WebForm1.aspx`, pulse el botón Ver código. Verá aparecer el módulo `WebForm1.aspx.vb`, un módulo en el que se define una clase escrita en lenguaje Visual Basic .NET. La cabecera es la siguiente:

```
Public Class WebForm1
    Inherits System.Web.UI.Page
```

La clase `WebForm1` está derivada de `System.Web.UI.Page` que, estableciendo una analogía, es como la clase `System.Windows.Forms.Form` pero para las aplicaciones Web. A su vez, la página definida en el documento `WebForm1.aspx` está derivada de esta clase `WebForm1`, de tal manera que hereda todos sus miembros. Sí, quizá pueda parecerle extraño, pero tenemos una página ASP.NET que hereda los miembros de una clase escrita en Visual Basic .NET. Esto es posible, lógicamente, porque los dos módulos, `WebForm1.aspx` y `WebForm1.aspx.vb`, generan código MSIL al ser compilados.

En el módulo Visual Basic, concretamente en la clase `WebForm1`, encontramos varios métodos más, entre ellos los encargados de gestionar los eventos `Load` e `Init`. Introduzca el código siguiente en el método `Page_Load()`:

```
Dim N As Byte
For N = 1 To 5
  Response.Write( _
      "<h" & N & ">Hola mundo</h" & N & ">")
Next
```

La propiedad `Response` es una de las que tiene este formulario Web que, en definitiva, es un objeto de la clase `Page`. Utilizamos el método `Write()` de dicha propiedad para introducir contenido en el documento cuando éste sea solicitado.

Si ejecuta el proyecto verá que se abre el cliente Web que tenga instalado en el sistema y que el resultado obtenido es, básicamente, el mismo que teníamos

con el ejemplo anterior. La diferencia está en que, en este caso, el código no forma parte del documento ASP.NET, por lo que resulta mucho más fácil mantenerlo.

Nota

Obviamente, el contenido introducido en la página desde el código Visual Basic .NET puede ser recuperado desde una base de datos u obtenido mediante algún otro proceso más complejo que el usado aquí como ejemplo.

Abra la carpeta correspondiente a este ejemplo y compruebe que, además del documento ASP.NET, alojado en el archivo `aspx`, existe una subcarpeta, llamada `bin`, en la que encontrará un módulo en forma de biblioteca de enlace dinámico. Ese módulo, en este caso llamado `WebSimple.dll`, es el que contiene el código ya compilado que utilizará el servidor para generar el contenido dinámico.

El diseñador de formularios Web

Al igual que para las aplicaciones Windows, Visual Basic .NET dispone de un diseñador específico para el trabajo con formularios Web, diseñador que aparece en la figura 11.4. En este caso el contenedor es la propia página Web. Ésta aparece como un fondo blanco con una cuadrícula sobre él, aunque, a diferencia de los formularios Windows, no tiene una barra de título, un borde o botones para maximizar, minimizar y cerrar, puesto que éstos no son elementos propios de una página HTML.

Los componentes se encuentran en el Cuadro de herramientas, concretamente en las páginas **Web Forms** y **HTML**. Los primeros se asemejan mucho a los controles usados en formularios Windows, hasta el punto de llamarse igual y tener en común la mayoría de las propiedades, métodos y eventos. El funcionamiento del control, como puede imaginar, es muy distinto ya que, en lugar de dibujarse en una ventana Windows, lo que tiene que hacer es generar el código HTML necesario para aparecer en un cliente web tipo Internet Explorer, Netscape Navigator, Opera, Mozilla o similar.

Más adelante, en este mismo capítulo, conocerá algunos controles para formularios Web, especialmente algunos específicos que no tienen un equivalente en los formularios Windows como son los de validación.

La clase `Page`

Si el contenedor típico en una aplicación Windows es un objeto de la clase `System.Windows.Forms.Form` o derivada, en el caso de las aplicaciones basadas en formularios web dicha clase es `System.Web.UI.Page`, es decir, la

clase `Page` alojada en el ámbito `System.Web.UI`, en el que también encontraremos la mayoría de controles para usar en este tipo de proyectos.

Al usar el diseñador con que cuenta Visual Basic .NET generamos automáticamente, según hemos visto antes, una clase derivada de `Page`. Esta clase hereda, por tanto, todos los métodos, propiedades y eventos definidos en dicha clase, muchos de los cuales son equivalentes a los que existían en ASP, como es el caso de `Request` y `Response`.

Uno de los primeros pasos que daremos, justo después de crear una nueva página, será modificar la propiedad `ClientTarget`, para lo cual recurriremos, como es habitual, a la ventana Propiedades. Con esa propiedad elegimos el cliente destinatario de la interfaz. Tal y como se aprecia en la figura 11.5 en principio tenemos tres posibilidades. Dependiendo de la selección efectuada ASP.NET generará el contenido del documento usando unas posibilidades u otras. Si el destinatario es Internet Explorer 5.0 se usará DHTML, por poner un ejemplo, mientras que si hemos elegido Internet Explorer 3.02 / Navigator 3.0 se utilizará HTML sin más.

Figura 11.5. Podemos seleccionar el tipo de cliente al que va destinado el proyecto

En caso de que estemos desarrollando un proyecto para una empresa es más fácil tener un control sobre el cliente web que utilizarán los usuarios de la aplicación, pero en caso contrario no debemos asumir que todo el mundo dispone de la última versión de Internet Explorer y, por tanto, puede usar DHTML y posibilidades similares.

Seleccionando un cliente más antiguo se obtendrá un resultado relativamente más pobre, en cuanto apariencia, pero, a cambio, podremos llegar a más clientes sin problemas.

Distribución del contenido

Al desarrollar aplicaciones basadas en formularios estamos acostumbrados a colocar los controles en posiciones absolutas, expresadas en puntos, ya que, hasta cierto punto, suponemos la configuración que tendrá el destinatario de esa aplicación.

Cuando se diseñan formularios web, por el contrario, no siempre puede hacerse esa suposición. De hecho, si alguna vez ha escrito código HTML sabrá que los elementos no se colocan habitualmente en una posición absoluta, sino que siguen un cierto flujo.

Cuando se crea un nuevo formulario web con Visual Basic .NET automáticamente se establece una distribución, conocida como `GridLayout`, que nos permite colocar los controles en cualquier punto de la página, exactamente igual que si se tratase de una ventana Windows. Este posicionamiento absoluto resulta cómodo pero, dependiendo del cliente, puede generar problemas de visualización. En este caso modificaríamos la propiedad `pageLayout` de la clase `Page` seleccionando la opción `PageLayout`. Al hacerlo verá que los controles se ajustan automáticamente a las dimensiones de la página y se colocan automáticamente donde convenga, sin una posición fija.

Utilizando la distribución `PageLayout` tendrá más dificultades para componer la interfaz de usuario si no está acostumbrado a trabajar con HTML. Tendrá que usar controles adicionales para compensar la falta de una posición absoluta pero, a cambio, su interfaz no estará limitada a Internet Explorer.

Eventos y gestores

Como puede ver, el diseño de la interfaz de usuario en un formulario Web no difiere demasiado respecto a los formularios Windows, y lo mismo ocurre con la asociación de gestores de eventos con el código que debe ejecutarse cuando los controles generen ciertas señales. Si inserta un botón en un formulario Web y hace doble clic sobre él verá que se abre el controlador correspondiente al evento por defecto. La asociación se establece declarando la variable que contiene la referencia al control con `WithEvents` y añadiendo la cláusula `Handles` al método que va a gestionarlo.

Encontrándonos en el editor de código, no tenemos más que desplegar las listas que hay en la parte superior para acceder a los controles que haya en la página y, a continuación, seleccionar el evento que deseamos controlar, exactamente igual que en una aplicación Windows.

A pesar de esta similitud aparente, la gestión de los eventos en una aplicación ASP.NET difiere considerablemente de la efectuada en un programa Windows. En éste el sistema operativo genera señales que transfiere directamente a la aplicación, mientras que en el caso de una aplicación Web los eventos se generan en un cliente remoto, posiblemente a miles de kilómetros, y son transferidos hasta el servidor web que, en definitiva, es el que ejecuta las reglas de negocio, en este caso los gestores de esos eventos.

Componentes HTML

Una de las funciones del diseñador de formularios Windows es servirnos como editor HTML, al estilo de FrontPage y productos similares. Para ello nos ofrece una serie de controles, en la página HTML del Cuadro de herramientas, así como una barra de botones específica (véase figura 11.6) que tendremos que activar usando el menú emergente de las barras de botones. Al trabajar con el diseñador de formularios Web también verá que aparecen opciones adicionales en el menú de Visual Basic .NET, concretamente tres nuevos menús: Tabla, Insertar y Marcos que facilitan el trabajo con tablas y marcos HTML, así como la inserción de imágenes y secciones.

Figura 11.6. Paleta de botones para edición de HTML

Los componentes HTML que insertemos en el formulario aparecen en el diseñador como el reflejo del código HTML que hay en la página, pero sin activar conexión alguna entre el cliente y el servidor. Es decir, estos componentes son los que utilizaríamos para establecer el contenido estático de la página, un contenido que se almacenará, sin más, en el documento `aspx` y, como tal, será transferido al cliente cuando éste efectúe una solicitud.

Aunque puede arrastrar cualquier componente de la página HTML al interior de la página y, a continuación, usar la ventana Propiedades para personalizarlo, verá que las propiedades difieren sustancialmente de las que hemos conocido en el capítulo previo para controles equivalentes. Puede verlo simplemente insertando en una página un `Button` de la página Web Forms y el mismo elemento de la página HTML. Al elegirlos podrá ver el conjunto de propiedades tan distinto con que cuentan.

Por último, los controles HTML no generan eventos que podamos controlar con gestores, según el esquema que ya conocemos. Si pretendemos responder a los eventos de los controles de una página deberemos recurrir a los de la página Web Forms, también conocidos como controles de servidor, y no a los de la página HTML.

Componentes de servidor

A la hora crear una aplicación basada en formularios Windows utilizamos, como se vio en el capítulo previo, unos controles específicos para este tipo de aplicaciones. Son controles que saben cómo generar su interfaz en el interior de una ventana Windows y cómo interactuar con el sistema.

Cuando la interfaz de usuario no es una aplicación nativa sino una página Web, como las de cualquier sede que pueda encontrarse en Internet, los controles que se utilizan son de tipo HTML. Estos se incluyen en un formulario

HTML y cuentan con una funcionalidad limitada, especialmente si se les compara con los controles típicos de una aplicación Windows. De ahí que a las interfaces nativas, basadas en formularios Windows, se las conozca como interfaces *ricas* en funcionalidad.

En ASP.NET se introducen los controles de servidor, un gran conjunto de componentes que podemos usar en cualquier documento ASP.NET y que, en el momento de solicitarse las páginas por parte del cliente, generan automáticamente código HTML. Es decir, se sigue manteniendo la compatibilidad con los clientes al tiempo que se facilita y amplía el campo de trabajo del diseñador/programador.

Al igual que los componentes para formularios Windows que pudimos conocer en un capítulo previo, los controles Web de servidor disponen de propiedades, métodos y eventos, muchos de ellos comunes. El ámbito con nombre `System.Web.UI.WebControls` contiene todas las clases de componentes de servidor que podemos usar en una página ASP.NET. Si consulta la documentación podrá ver que tenemos desde los típicos botones, listas y cajas de selección hasta cuadrículas de datos o calendarios.

Uso de componentes de servidor

Inicie un nuevo proyecto basado en formularios Web y observe la lista de controles existentes en la página Web Forms del Cuadro de herramientas, página que aparece abierta por defecto. Por una parte encontrará muchos de los controles que ya conoce de Windows: cajas de texto, botones, listas, etc. En realidad son controles de servidor ASP.NET con una funcionalidad equivalente a los que conoce, no son los mismos controles. Más abajo encontrará componentes específicos, como los que permiten efectuar validaciones del contenido de controles.

Con el objetivo de realizar una prueba simple tome del Cuadro de herramientas un control `Calendar` y un `TextBox`, insertándolos en la página tal y como se aprecia en la figura 11.7. El aspecto inicial del calendario no es el que puede verse en esta figura, para conseguir dicha apariencia se han modificado algunas propiedades: `BackColor`, para establecer el color de fondo general; `DayHeaderStyle`, configurando la cabecera donde aparecen los nombres de los días; `SelectedDayStyle`, fijando los atributos del día elegido, etc.

Nos interesa que, cuando el cliente solicite el documento y seleccione un día, el día elegido aparezca en el interior de la caja de texto. Con este fin, teniendo seleccionado el calendario, hacemos doble clic sobre él para generar el método asociado al evento `SelectionChanged`. En su interior introducimos la siguiente sentencia:

```
TextBox1.Text = "Has elegido la fecha " & _
                Calendar1.SelectedDate
```

Asignamos a la propiedad `Text` de la caja de texto un mensaje seguido del contenido de la propiedad `SelectedDate` del calendario. Dicha propiedad contiene la fecha seleccionada en ese instante en el calendario.

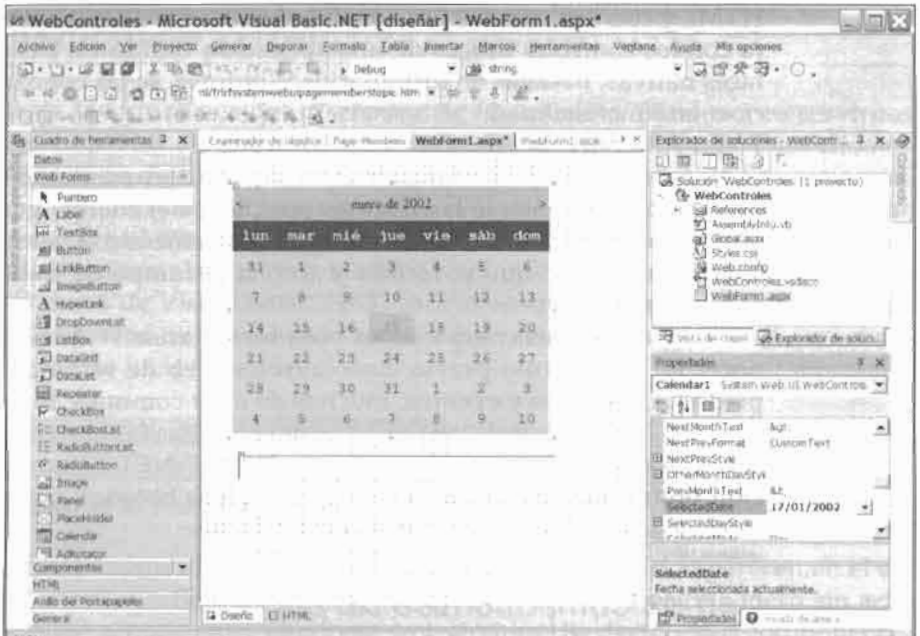


Figura 11.7. Diseño de la página con el calendario y la caja de texto

Nota

Observe que la forma de trabajar con los componentes de servidor ASP.NET es idéntica a la que conocimos previamente para los controles Windows. Basta una asignación a una propiedad, por ejemplo, para cambiar el título o contenido de un control. Detrás de esa asignación, no obstante, el proceso que lleva a cabo el control es muy distinto, como podrá imaginar. Para cambiar el título de un control Windows basta con enviarle un mensaje a la correspondiente ventana, mientras que en el caso de los controles de servidor se establece una comunicación entre un cliente y un servidor Web conectados a través de una red.

Si ejecuta el proyecto verá que el comportamiento es el esperado (véase figura 11.8). Aparece una página HTML con un calendario y una caja de texto. Al seleccionar una fecha, ésta aparece en la caja de texto.

Examen del código HTML en el servidor

Sabemos que, al trabajar con un formulario Windows, cualquier modificación de propiedades, inserción de componentes o uso de un evento se traduce en la generación de código que es, automáticamente, introducido en el correspondiente módulo. Cuando trabajamos con una aplicación Web ASP.NET te-

nemos dos módulos que van actualizándose: el módulo de código y la propia página ASP.NET.

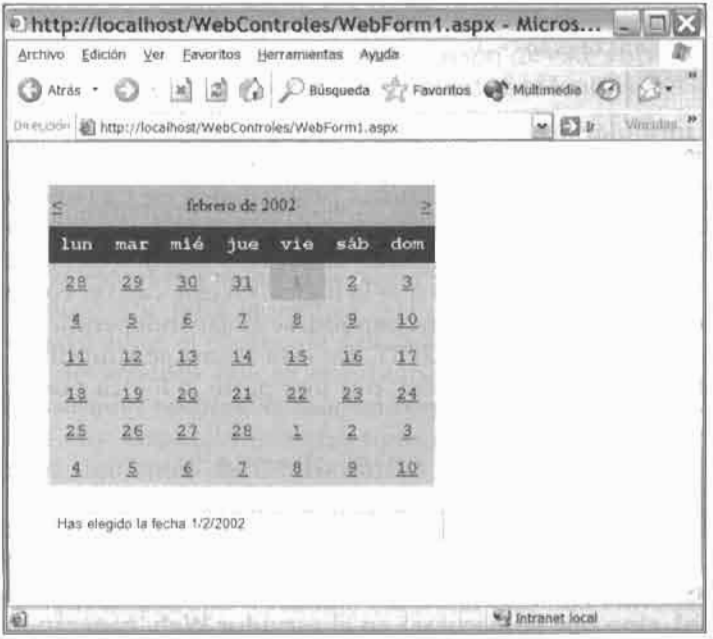


Figura 11.8. La aplicación Web en funcionamiento en Internet Explorer

En el módulo de código, en este caso `WebForm1.aspx.vb`, verá que se añade a la clase derivada de `Page` un miembro por cada control insertado en la página. En el método `InitializeComponent()` se conectan los eventos de los controles con métodos de la clase, en este ejemplo concreto el evento `SelectionChanged` del calendario se conecta a un método en el cual hemos introducido la asignación del punto anterior. No encontramos en este módulo, sin embargo, las asignaciones hechas a propiedades.

Si nos ponemos a examinar el código HTML de la página ASP.NET, el módulo `WebForm1.aspx`, encontraremos, aparte de los encabezados que vimos anteriormente, un código similar al siguiente:

```
<form id="Form1" method="post" runat="server">
   <asp:Calendar id="Calendar1"
       style="LEFT: 72px; TOP: 57px"
        runat="server" Width="291px"
        Height="270px" ForeColor="Black"
        BackColor="#FFC0C0"
        SelectedDate="2001-10-14">
        <DayHeaderStyle Font-Names="Courier New"
             Font-Bold="True" ForeColor="White"
             BackColor="#0000C0">
        </DayHeaderStyle>
        <SelectedDayStyle Font-Bold="True"
             ForeColor="#C00000">
```

```
        </SelectedDayStyle>
    </asp:Calendar>
    <asp:TextBox id="TextBox1"
        style="LEFT: 73px; TOP: 347px"
        runat="server" Width="292px" Height="22px">
    </asp:TextBox>
</form>
```

El formulario ASP.NET contiene dos componentes: `Calendar` y `TextBox`. Observe cómo se establecen las propiedades de cada uno de ellos, como posición, nombre, color, etc. Algunas propiedades de `Calendar` constan de otras propiedades, es lo que ocurre con `DayheaderStyle` y `SelectedDayStyle`.

Las propiedades, que afectan al aspecto estético del resultado, están en la página ASP.NET, mientras que la lógica, en este caso el código del método asociado al método `SelectionChanged`, se alojan independientemente en el módulo de código Visual Basic .NET. De esta forma se simplifica el mantenimiento de ambos elementos: interfaz, por una parte, y lógica por otra.

Examen del código en el cliente

Si observa el código anterior comprobará que el formulario y los componentes cuentan con un atributo, `runat`, que indica que el proceso debe efectuarse en el servidor. Esto significa que ese código no será enviado al cliente tal cual, sino que se ejecutará en el servidor Web, generando un nuevo código que, en último término, será el que reciba Internet Explorer o el cliente que estemos usando.

Puede ejecutar el proyecto y, una vez se haya abierto la página, utilizar la opción adecuada del cliente para acceder al código obtenido. Verá que no existen controles ASP.NET, sino controles HTML y código JavaScript. Básicamente se utiliza una tabla HTML para componer el calendario, una tabla en la que cada celdilla, representando un día del mes, es un enlace a un procedimiento JavaScript que se encarga de enviar el formulario al servidor con el parámetro apropiado, momento en el que ASP.NET ejecutará la lógica, en nuestro módulo Visual Basic .NET, y devolverá la respuesta al cliente.

Modificando algunas propiedades de la página, en la ventana Propiedades, es posible adaptar el resultado que generan los componentes de servidor para conseguir un mejor comportamiento en ciertos tipos de clientes. También podemos seleccionar el lenguaje de guiones por defecto del cliente.

Componentes de validación

Los clientes web que se usan actualmente, como los ya citados Internet Explorer y Netscape Navigator, se ejecutan sobre sistemas que tienen una cierta capacidad de proceso. Esto significa que algunas de las tareas que con ASP.NET efectuamos en el servidor podrían, asimismo, ser efectuadas en el servidor. La virtud estará en encontrar el término medio entre el reparto de tareas, cliente versus servidor, y el mejor aprovechamiento del ancho de banda.

Cuando se utilizan formularios web para solicitar datos al usuario hay una tarea indispensable: la validación de los datos. Ésta podría efectuarse en el servidor que, al recibir los datos, los comprobaría y notificaría al cliente cualquier error. El problema en este planteamiento es que cada corrección del usuario debe viajar hasta el servidor, y cada notificación de éste volver de nuevo al cliente. Si los errores son varios el proceso puede ser lento si no se utiliza una red propia o lo suficientemente rápida.

La alternativa, está claro, es que ese proceso de validación se efectúe completamente en el propio cliente, de tal manera que el formulario sólo se envíe al servidor cuando toda la información que contiene sea correcta. Para ello bastaría con escribir algo de código embebido en el propio documento ASP.NET, código que se ejecutaría no en el servidor sino en el cliente.

Utilizando ASP.NET la ventaja está en que no tendremos que ser nosotros los que escribamos el código de validación, ya que existe un conjunto de componentes de servidor que se encargan de generarlo automáticamente. Estos componentes, no visibles, se asocian con los controles de entrada de datos. Básicamente hay disponibles cuatro tipos de validaciones distintas, para la cual existen los siguientes cuatro controles:

- `RequiredFieldValidator` – Se utiliza para asegurar que un campo obligatorio no queda en blanco al enviar el formulario.
- `RangeValidator` – Permite controlar que una entrada se encuentre entre un cierto rango de valores.
- `CompareValidator` – Similar al anterior, en este caso podemos utilizar un operador relacional para verificar que el valor es menor, mayor, igual o distinto a una cierta propiedad u otro valor.
- `RegularExpressionValidator` – Lo podemos utilizar para conseguir que la entrada de un dato se ajuste a una cierta plantilla o expresión, por ejemplo para la entrada de códigos postales o teléfonos.

A estas cuatro posibilidades hay que añadir una quinta, asociada al componente `CustomValidator`, que es la mas flexible de todas ya que nos permite asociar un método a la comprobación de datos.

Veamos un sencillo ejemplo de uso de estos componentes, concretamente de `RequiredFieldValidator`. Iniciamos una nueva aplicación Web e introducimos en la página una etiqueta, una caja de texto, un botón y hacemos doble clic sobre éste introduciendo el código mostrado a continuación:

```
TextBox1.Text = "Hola " & TextBox1.Text
```

Nada especialmente sorprendente, tan sólo utilizamos la caja de texto para mostrar un saludo seguido del nombre que se haya introducido.

Añadimos a la página un componente `RequiredFieldValidator` que, en principio, aparecerá como un mensaje en color rojo. Modificamos su propiedad `ErrorMessage`, estableciendo un mensaje explicativo de cuál es el problema, y también la propiedad `ControlToValidate` facilitando la referencia al

control cuyo contenido tiene que validarse, en este caso la caja de texto. En la figura 11.9 se puede observar la página, durante la fase de diseño, con el `RequiredFieldValidator` seleccionado y sus propiedades en la parte inferior derecha.

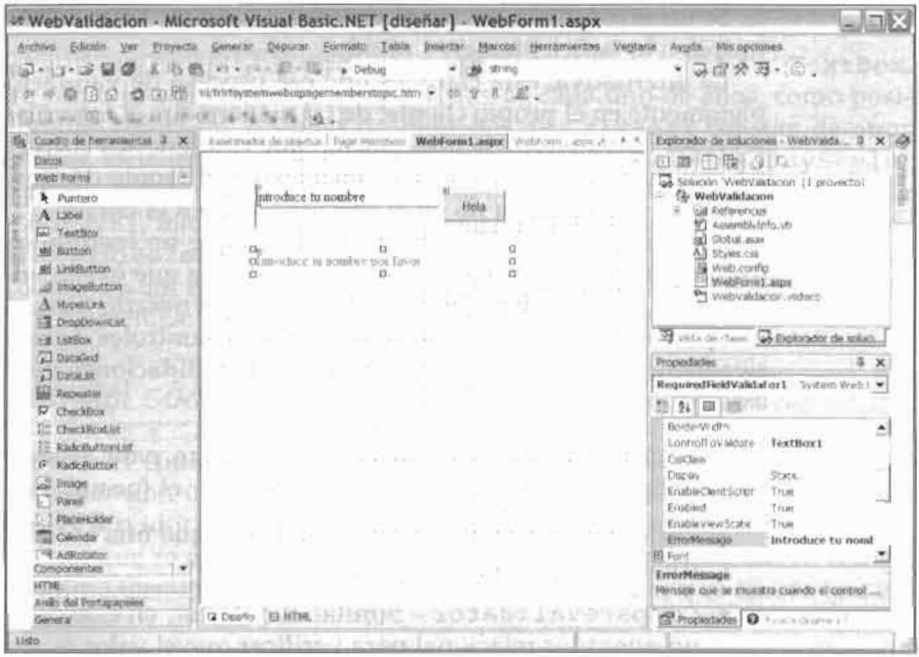


Figura 11.9. Página con el componente de validación asociado a la caja de texto

Al ejecutar el proyecto verá aparecer la página con la caja de texto y el botón. Si introduce un dato y pulsa el botón todo funcionará como es de esperar, pero si pulsa el botón dejando la caja de texto vacía aparecerá el mensaje de error. La comprobación de que la caja de texto está vacía se efectúa en el cliente, sin necesidad de enviar el formulario al servidor, gracias a un guión en JavaScript que puede encontrar en el código fuente recibido por el cliente.

Solicitudes y respuestas

Una aplicación web basada en formularios ASP.NET, como los descritos en este capítulo, es ejecutada por el servidor web al recibir una solicitud de un cliente. Cada formulario web creado se representa mediante un objeto de la clase `Page`, tal y como se indicó antes, clase en la que encontramos propiedades con información sobre esa solicitud y que habilitan el envío de la respuesta. En el primer ejemplo del capítulo, por ejemplo, utilizamos la propiedad `Response`, que hemos heredado de `Page` al estar nuestro formulario derivado

de dicha clase, para enviar una respuesta y la propiedad `Request` para recuperar datos de la solicitud.

La propiedad `Request` de la clase `Page` mantiene una referencia a un objeto de la clase `HttpRequest`. Ésta dispone de un importante número de miembros que, en su mayor parte, podemos utilizar para recuperar información relativa a la propia solicitud, el cliente que la ha hecho o su configuración de software.

De manera análoga, la propiedad `Response` contiene una referencia a un objeto de la clase `HttpResponse`. Podemos usar sus miembros para enviar una respuesta al cliente y controlar diversos aspectos del contenido, como el plazo de expiración o las *cookies* asociadas.

Identificación del cliente

En ocasiones, con el fin de personalizar la respuesta enviada al cliente, puede interesarnos identificarlo, hasta cierto punto, usando para ello la información que su cliente nos facilita en la solicitud. Toda esa información se almacena en el servidor en una serie de variables de entorno. Algunos de los datos podemos recuperarlos directamente de propiedades del objeto `HttpRequest`, mientras que otros precisarán la lectura directa de las mencionadas variables.

Algunas propiedades de `HttpRequest` mantienen directamente una cadena de caracteres con un dato concreto. Es el caso de `UserAgent`, `UserHostName` o `UserHostAddress` que contienen la cadena de identificación del cliente, el nombre del ordenador del cliente y su dirección IP. Otras, como es el caso de `Browser`, lo que tienen es una referencia a un objeto, en dicho caso de tipo `HttpBrowserCapabilities`, que, a su vez, disponen de multitud de propiedades informativas. Mediante la propiedad `Browser` podemos recuperar datos como el nombre del cliente, su versión, la descripción del sistema en el que se ejecuta, si acepta o no marcos, tablas o el uso de los lenguajes JavaScript o VBScript. Son datos todos ellos que pueden ayudarnos a ajustar el código enviado al cliente, aunque ésta es una tarea de la que, usando formularios Web, se ocupa el propio motor de ASP.NET.

La propiedad `ServerVariables` de `HttpRequest` es una colección de elementos compuestos de una clave y un valor. Este tipo de colecciones se gestionan con la clase `NameValueCollection`. La clave de cada elemento es el nombre de una variable de servidor, por ejemplo `HTTP_ACCEPT`, `REMOTE_HOST` o `SERVER_NAME`. Podemos usar la propiedad `ServerVariables` como si de un arreglo se tratase, usando el nombre de la variable como índice a fin de recuperar el valor asociado.

No tiene mucho sentido enumerar aquí las propiedades de `HttpRequest`, `HttpBrowserCapabilities` y las variables de servidor existentes, ya que es una información que puede encontrar rápidamente en la ayuda electrónica de Visual Basic .NET. Lo importante es que sepamos que esos datos existen y que podemos aprovecharlos en caso necesario. Si quiere realizar una prueba de los pasos siguientes: inicie un nuevo proyecto de aplicación Web ASP.NET, haga doble clic sobre el fondo de la página e inserte el código siguiente:

```
Private Sub Page_Load(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles MyBase.Load

  With Request.Browser
    Response.Write( _
        "El cliente web es: <b>" & .Type & " - " & .Version _
        & "</b><br>" & _
        "El sistema operativo es: <b>" & .Platform & "</b><br>" & _
        "El cliente contempla el uso de marcos: <b>" & .Frames _
        & "</b><br>" & _
        "El cliente acepta <i>cookies</i>: <b>" & .Cookies _
        & "</b><p>")
  End With


  Dim Clave As String
  Response.Write("<pre>")
  For Each Clave In Request.ServerVariables
    Response.Write(Clave & "=" & _
        Request.ServerVariables(Clave) & "<br>")
  Next
End Sub
```

Como puede ver, usamos algunas de las propiedades de la clase `HttpBrowserCapabilities` para identificar al cliente, mostrando su nombre y la plataforma, así como la indicación de si contempla el uso de marcos y tablas. Después recorremos la colección de variables de servidor mostrando todas las claves y sus valores asociados.

El resultado es el que puede verse en la figura 11.10, no útil por sí pero muy informativo.

Nota

En lugar de utilizar los datos de la solicitud para simplemente devolverla al cliente, como hemos hecho en este ejemplo, podríamos almacenar parte de la información en una base de datos del servidor para, por ejemplo, tener una estadística de los clientes y plataformas de los usuarios que acceden a nuestra aplicación lo cual, en definitiva, nos permitirá ajustar mejor el servicio.

Parámetros de la solicitud

Al enviar la solicitud al servidor, además del camino y el nombre del formulario, el archivo `aspx`, el cliente puede añadir parámetros adicionales. En realidad éstos suelen añadirse automáticamente ante determinadas acciones del cliente, como la selección de un enlace o el envío de un formulario de datos. Lo que nos interesa a nosotros es saber cómo podemos recuperar esos parámetros desde el servidor.

Figura 11.10. Alguna información sobre el cliente que ha enviado la solicitud

Suponga que tiene un formulario inicial con varios enlaces, como el mostrado en la figura 11.11. Hemos insertado dos componentes `HyperLink`, asignando a la propiedad `NavigateUrl` los valores `WebForm1.aspx?Compras` y `WebForm1.aspx?Catalogo`, respectivamente. El carácter ? separa el nombre del documento a solicitar de la lista de parámetros, en este caso tan sólo uno. Si existiesen varios los separaríamos entre sí con el carácter `&`.

Si ejecutamos el proyecto tal como está ahora verá aparecer la página con los dos enlaces pero, al pulsar sobre ellos, simplemente se recibirá de nuevo el mismo documento. Los parámetros están viajando hasta el servidor, pero éste no los está utilizando.

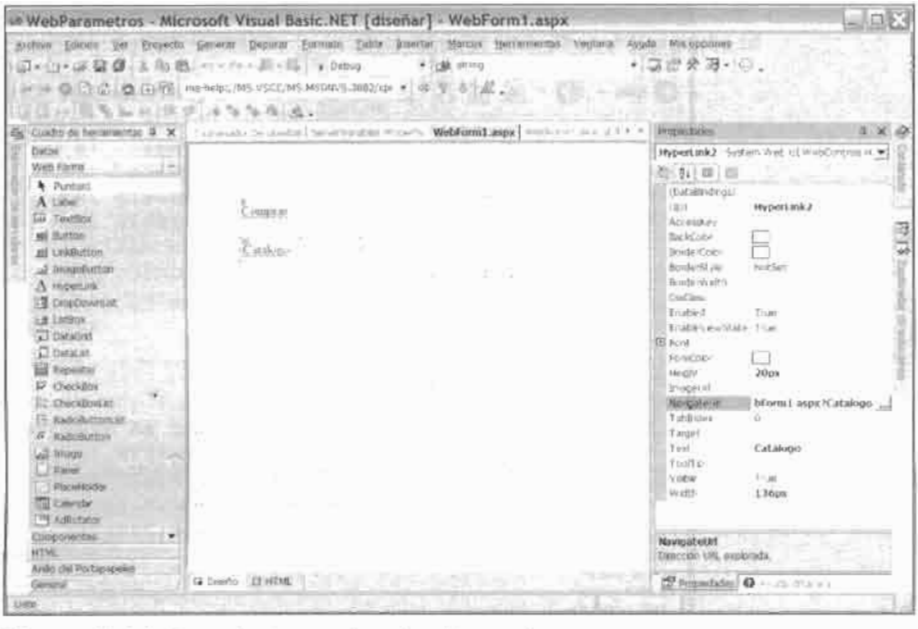


Figura 11.11. Formulario con los dos hiperenlaces

Para recuperar la lista de parámetros adjuntos a una solicitud se usa la propiedad `QueryString` de la clase `HttpRequest`. Esa propiedad contiene una referencia a un objeto `NameValueCollection`, como `ServerVariables`. Mediante la propiedad `Count` de la colección podemos saber cuántos parámetros se han facilitado, utilizando dicha información para acceder a ellos mediante un índice numérico.

Hacemos doble clic sobre el fondo de la página, a fin de acceder al método que actúa como gestor del evento `Load`, e introducimos el código siguiente. Como puede ver es muy simple: comprobamos si hay al menos un parámetro y, a continuación, lo devolvemos al cliente:

```
If Request.QueryString.Count <> 0 Then
  Response.Write(Request.QueryString(0))
End If
```

Ahora al solicitar el documento obtendrá, como antes, el formulario con los dos enlaces. Al pulsar sobre cualquiera de ellos, sin embargo, verá la respuesta. Incluso puede editar directamente el URI para introducir cualquier parámetro no preestablecido. En la práctica no devolveríamos el parámetro al cliente sino que lo usaríamos para elaborar la respuesta adecuada.

Parámetros de formularios

Al insertar controles en un formulario Web, indistintamente de que sean controles de servidor o controles HTML, el diseñador genera automáticamente

un formulario HTML usando la etiqueta `<form>`. Este tipo de formularios generalmente cuentan con un botón que se encarga de transferir la información al servidor, para lo cual puede utilizar dos métodos distintos: `POST` y `GET`. El segundo es el que hemos utilizado en el punto anterior para enviar los parámetros al servidor.

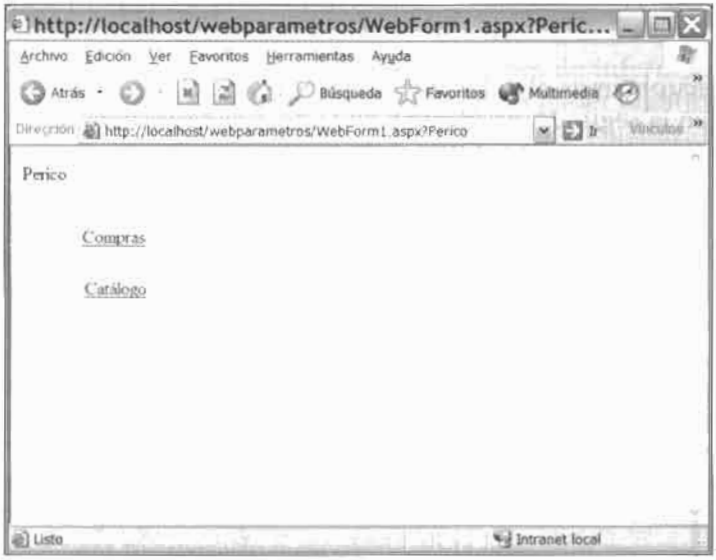


Figura 11.12. La aplicación responde devolviendo el parámetro entregado en el URL

Para enviar el contenido de un formulario por defecto se utiliza el primer método, `POST`, consistente en incluir los datos en el cuerpo del mensaje HTTP, en lugar de hacerlo en el propio URL. De esta manera se oculta de los ojos del usuario y la comunicación no resulta tan obvia. Si accede al código HTML de un formulario Web verá que la etiqueta con la que se abre el formulario es la siguiente:

```
<form id="Form1" method="post" runat="server">
```

La diferencia, sin embargo, no es sólo estética. Los datos de un formulario enviado con el método `GET` pueden recuperarse mediante la propiedad `QueryString` que ya conocemos. Por el contrario, si se utiliza el método `POST` dicha propiedad no tendrá contenido, tendremos que usar en su lugar la propiedad `Form`. También se trata de una colección, pudiendo utilizarse como índice el nombre del control cuyo valor deseamos recuperar.

Suponiendo que hemos insertado en un formulario una caja de texto, un control `CheckBox`, un botón y una segunda caja de texto en la parte inferior, tal y como puede verse en la figura 11.13, haríamos doble clic sobre el botón y usaríamos el código siguiente para comprobar el funcionamiento de la colección `Form`.

```
Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
  TextBox2.Text = "Hola " & Request.Form("TextBox1") & ","
  If Request.Form("CheckBox1") = "on" Then
    TextBox2.Text &= "me alegro de que tengas una vivienda propia"
  Else
    TextBox2.Text &= "siento que no tengas vivienda propia"
  End If
End Sub
```

Recuperamos el valor introducido en la primera caja de texto y lo mostramos en la segunda, añadiendo un texto que dependerá de que se haya o no activado el control CheckBox. Éste tomará el valor "on" si se ha marcado.

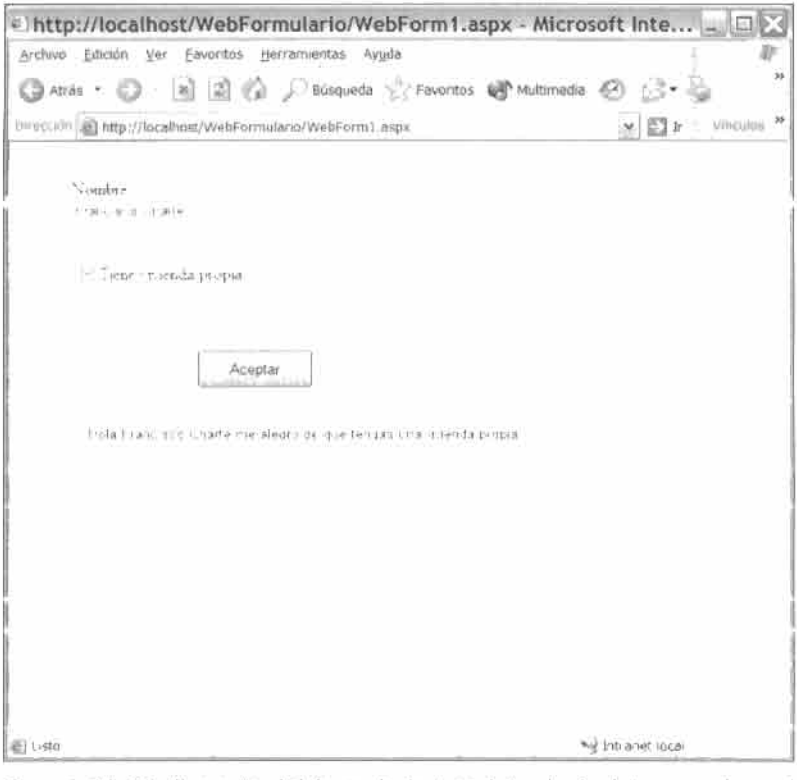


Figura 11.13. Aspecto del formulario tras introducir datos y pulsar el botón

Nota

Si modifica el método usado para enviar la información, cambiando POST por GET, y ejecuta de nuevo la aplicación verá que los datos se transmiten a través del URL. En cualquier caso, y asumiendo que estamos utilizando controles de servidor, siempre será más fácil usar directamente `TextBox1.Text` que `Request.Form("TextBox1")` para recuperar el texto

introducido en una caja de texto. La clase `Form`, por el contrario, la necesitaríamos si los controles utilizados son de tipo HTML, sin conexión con el servidor.

Uso de cookies

Se conoce como *cookies* a pequeñas porciones de información que el servidor almacena en los clientes, generalmente para poder identificarlos y, en cierta forma, personalizar la interfaz para hacerla más amigable. El cliente es libre de aceptar o denegar la operación de escritura en su sistema, al igual que el servidor lo es de requerir al cliente esa posibilidad si desea aprovechar ciertas opciones de la aplicación sede.

Las clases `HttpRequest` y `HttpResponse` disponen de una propiedad, llamada `Cookies`, que contiene una colección de objetos `HttpCookie`. En el caso de `HttpRequest` se trata de las *cookies* recuperadas del cliente en el momento en que éste hacía la solicitud, mientras que `HttpResponse` alojará las que deseemos escribir al enviar la respuesta. Cada objeto `HttpCookie` cuenta con una serie de propiedades que permiten recuperar, o establecer, el valor a almacenar en el cliente, la fecha de caducidad o el camino virtual asociado al valor. También se puede almacenar múltiples valores asociados a una misma clave.

Por regla general se asocia un nombre a cada *cookie*, nombre que se usa como índice para recuperarlo o escribirlo en la colección representada por la mencionada propiedad `Cookies`. El código siguiente, asociado al evento `Load` de un formulario ASP.NET, identifica a cada usuario individual que acceda a la página indicándole si es la primera vez que la solicita o bien mostrándole la fecha y hora de su último acceso.

```
Private Sub Page_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

  If Request.Cookies("UltimaVisita") Is Nothing Then

    Response.Write("Es la primera vez que visitas esta web")
   Else
    Response.Write("Tu ultima visita fue el " & _
                   Request.Cookies("UltimaVisita").Value)
  End If

  Response.Cookies("UltimaVisita").Value = Date.Now

  Response.Cookies("UltimaVisita").Expires = Date.Now.AddYears(1)
End Sub
```

En lugar de la fecha de la última visita, o además de este dato, nuestra aplicación podría solicitar al usuario, la primera vez que accediese, una serie de datos para personalizar la información que quiere ver en posteriores accesos. Es la técnica usada por algunos portales para permitir a los clientes configurarlo según sus propias preferencias.

Figura 11.14. Información mostrada al acceder una segunda vez a la página

Módulos de configuración

La configuración de las aplicaciones ASP.NET se efectúa mediante archivos con sintaxis XML, de tal forma que existen diferentes ramas y niveles jerárquicos con valores. Dichos archivos de configuración se llaman `web.config` y pueden alojarse en distintos puntos.

Existe una secuencia de búsqueda para que ASP.NET localice los archivos de configuración, estableciéndose un cierto mecanismo de herencia de tal forma que una aplicación hereda toda la configuración genérica de ASP.NET, la configuración personalizada de la aplicación e incluso una configuración local por carpetas.

El archivo de configuración general de ASP.NET puede encontrarlo en el interior de la carpeta `\WINDOWS\Microsoft.NET\Framework` de su sistema, en una subcarpeta cuyo nombre depende de la versión de la plataforma .NET. En mi caso concreto esa subcarpeta es `v1.0.2914`, y en ella está el archivo `machine.config`, conteniendo información general de configuración. Éste contiene las secciones necesarias para poder configurar la compilación de los documentos ASP.NET, detectar las posibilidades del cliente que está solicitándonos la información, gestionar diferentes protocolos de comunicación, recuperar distintas unidades de código, etc.

No es aconsejable manipular el contenido de este archivo y, realmente, tampoco es necesario.

Podemos crear nuestros propios archivos `web.config`, que copiaríamos en la carpeta raíz de nuestra aplicación o, incluso, en las subcarpetas específicas para cada componente de esa aplicación.

Edición del archivo web.config

Cada vez que creamos una aplicación Web, el asistente añade al proyecto un archivo web.config con una configuración por defecto. Para modificar su contenido basta con hacer doble clic sobre él en el Explorador de proyectos. El formato general de este archivo es el siguiente:

```
<configuration>
  <seccion>
    <valor>
    </valor>
  </seccion>
</configuration>
```

Las secciones pueden ser cualquiera de las predefinidas en el archivo de configuración global u otras que nosotros mismos definamos. Dependiendo de la sección, los valores posibles serán unos u otros.

Supongamos que hemos desactivado la apertura automática de unidades con código por parte de ASP.NET, para lo cual eliminariamos la línea <add assembly="*"/> del archivo de configuración general indicado antes. Esto causaría que las bibliotecas copiadas en la subcarpeta bin de nuestra aplicación no se abriesen automáticamente, de tal manera que algunas aplicaciones podrían no funcionar adecuadamente.

La solución, para establecer una configuración personalizada, sería crear un archivo web.config con el contenido siguiente, guardando el archivo en la carpeta raíz de nuestra aplicación.

```
<configuration>
    <compilation>
        <assemblies>
            <add assembly="MisClases"/>
        </assemblies>
    </compilation>
</configuration>
```

Con esto conseguimos que sólo se alojen en memoria las unidades de código que nosotros indiquemos explícitamente, en lugar de todas las que existan en la carpeta bin.

Almacenamiento de datos de aplicación

Como se ha indicado antes, existen una serie de secciones predefinidas para los archivos de configuración web.config. Dichas secciones están definidas en el archivo de configuración general. Una de esas secciones, llamada appSettings, puede ser usada para almacenar datos de configuración propios de la aplicación, en forma de parejas compuestas de una clave y un valor.

Partamos, una vez más, de un supuesto práctico. Imagine que su aplicación va a estar disponible en varios idiomas, existiendo caminos diferentes para los

documentos de cada uno de ellos. Si introdujésemos directamente esos caminos en los documentos estaríamos limitando las capacidades de ampliación y mantenimiento. Sería mucho más efectivo almacenar en el archivo de configuración los elementos necesarios que contuviesen los idiomas y las carpetas asociadas.

Para añadir datos de aplicación debemos crear el apartado `<appSettings>` que, por defecto, no existe en el archivo de configuración original. Creada la sección, añadiríamos las claves y sus valores mediante etiquetas `<add>`. El siguiente fragmento, por ejemplo, añade dos claves y dos valores. Añadir un nuevo idioma sería tan fácil como añadir una nueva línea en la sección `idiomas`.

```
<appSettings>
   <add key="Español" value="content\es" />
   <add key="English" value="content\en" />
</appSettings>
```

De forma análoga podríamos añadir, por ejemplo, valores para contener las cadenas de conexión a bases de datos, URLs de lugares externos, etc. El sistema sería prácticamente idéntico al anterior.

Recuperando información de configuración

Preparar un archivo de configuración como el anterior no nos serviría de nada si, posteriormente, no contásemos con algún sistema que nos permita recuperarla desde nuestro documento ASP.NET. Lógicamente dicho mecanismo existe, consistiendo, básicamente, en usar los métodos de un componente para recuperar la sección de configuración que nos interese.

El objeto que nos interesa se llama `Context` y está disponible para cualquier aplicación ASP.NET. Dicho objeto cuenta con un método, llamado `GetConfig()`, con el cual podemos recuperar una sección completa del archivo `web.config`. El nombre de dicha sección la facilitaríamos como único parámetro al método `GetConfig()`.

Lo que obtenemos como valor de retorno es un objeto genérico, clase `Object`, que deberemos convertir para poder utilizarlo. Convirtiéndolo a un objeto de la clase `NameValueCollection` dispondremos de una tabla a cuyos elementos, que serían los introducidos en la sección, accederíamos con el típico operador (). Lógicamente, el índice sería el nombre clave del valor en lugar de un número.

Una vez que hemos recuperado el valor asociado a la clave, podríamos utilizarlo según nos interese, ya sea para acceder a una base de datos, establecer el camino base para los documentos en un cierto idioma, etc.

Asumiendo que hemos creado el archivo de configuración citado en el punto previo, veamos en la práctica cómo podríamos permitir al cliente seleccionar un idioma, obteniendo el camino donde se encuentran los documentos correspondientes. Nosotros nos limitaremos a mostrar ese camino, aunque podríamos usarlo, como se ha dicho, para fijar la carpeta de acceso a los documentos. El siguiente sería parte del contenido del módulo `aspx` de la aplicación:

```
<script Language="vb" runat="server">
Public Sub CambioIdioma( _
        ByVal O As Object, ByVal e As EventArgs)
  Dim Coleccion As NameValueCollection = _
    Context.GetConfig("appSettings")
  Dim Valor As String = _
      Coleccion(Idioma.SelectedItem.Value)
  Response.Write(Valor)
End Sub

</script>
<body MS_POSITIONING="GridLayout">
  <form id="Form1" method="post" runat="server">
    <asp:dropdownlist id="Idioma" runat="server">
      <asp:listitem>Español</asp:listitem>
      <asp:listitem>English</asp:listitem>
    </asp:dropdownlist>
    <asp:button runat="server" Text="Consultar ..."
      OnClick="CambioIdioma" ID="Button1" />
  </form>
</body>
```

Hemos incluido en el formulario HTML un componente `dropdownlist`, es decir, una lista desplegable con los idiomas que tenemos como opción. También se ha insertado un botón asociando el evento que se produce al pulsarlo con el método que hay al principio. En él usamos la propiedad `SelectedItem` del control para recuperar una referencia al elemento seleccionado que, a su vez, cuenta con una propiedad llamada `Value` conteniendo el valor. En este caso la lógica se ha embebido en el propio documento ASP.NET en forma de guión, pero podríamos haberla introducido separadamente como hemos hecho en ejemplos previos.

Al solicitar este documento vería, en principio, la lista desplegable con los idiomas y el botón. Seleccione una de las opciones y después pulse el botón. Verá entonces aparecer, como muestra la figura 11.15, el camino indicado en el archivo `web.config`. Lógicamente, cualquier cambio en dicho archivo tendría una repercusión inmediata en el funcionamiento del documento ASP.NET, el cual mostraría el nuevo valor en cuanto otro cliente volviese a solicitar dicha página.

Puntos clave

- ASP.NET supone una mejora importante respecto a ASP, su predecesor, al dar libertad en la elección del lenguaje para escribir los guiones, compilar éstos en lugar de interpretarlos y hacer posible la separación entre código ejecutable y código HTML.
- Visual Basic .NET dispone de un diseñador que facilita la creación de formularios Web usando los mismos recursos que conocimos en el capítulo previo para los formularios Windows.

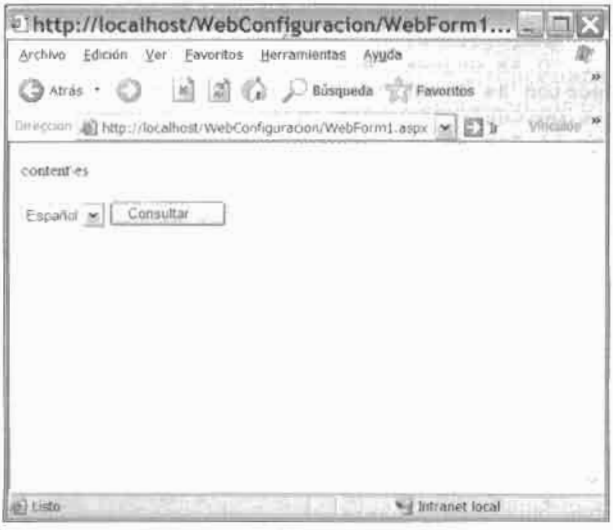


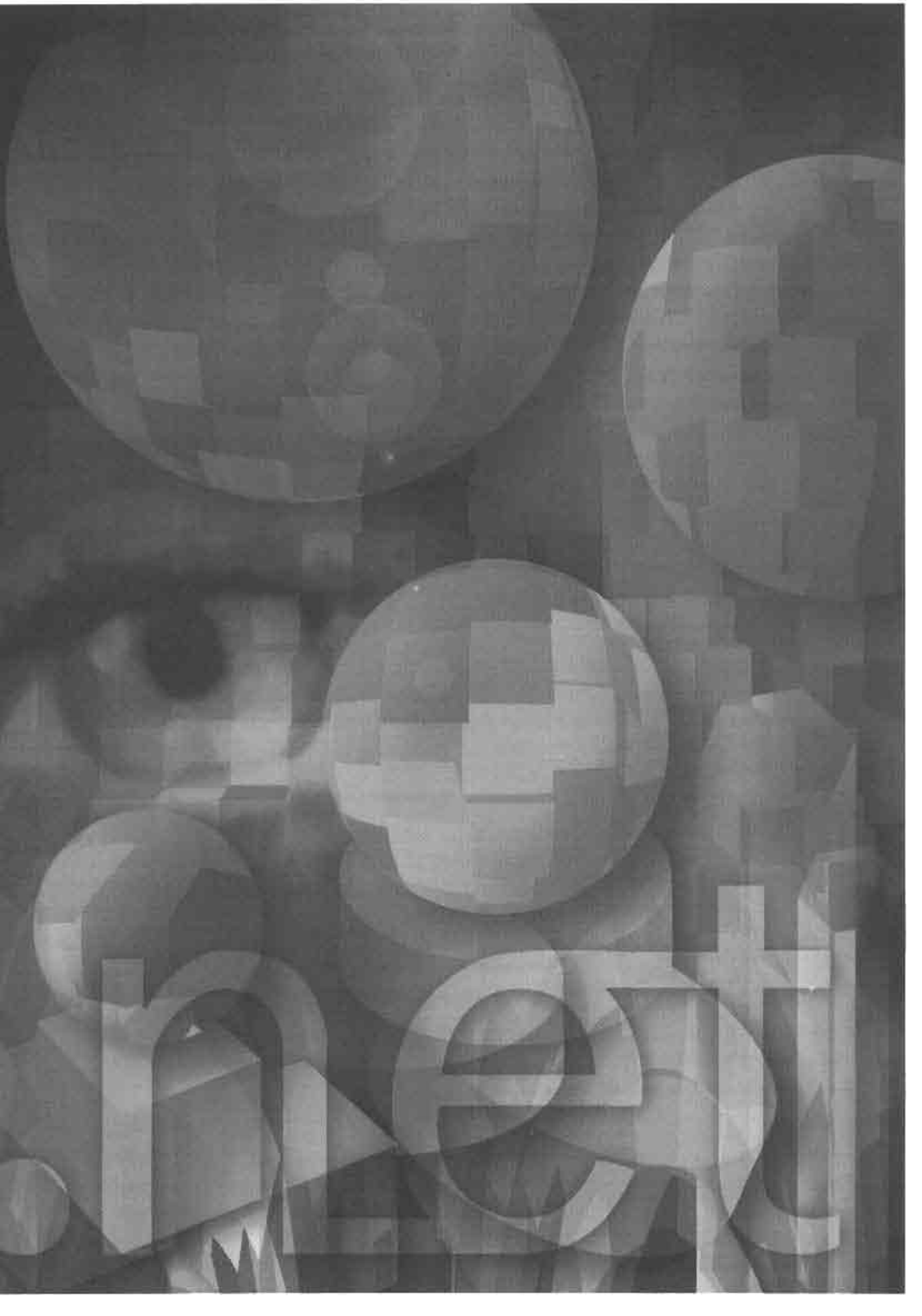
Figura 11.15. Al elegir un idioma el documento nos muestra el camino indicado en el archivo de configuración

- El contenedor de primer nivel, el formulario Web, es la representación de un objeto de la clase `Page`. Ésta dispone de propiedades que permiten adaptar la generación del contenido para distintos clientes.
- Un formulario web está compuesto de dos módulos. El primero, con extensión `aspx`, aloja el código HTML, mientras que el segundo, con extensión `aspx.vb`, contiene el código Visual Basic .NET con la lógica.
- Los componentes HTML sólo aparecen en el módulo `aspx`, enviándose al cliente como contenido estático.
- Los componentes de servidor aparecen en ambos módulos, estableciéndose un enlace que facilita la gestión en el servidor de los eventos que se produzcan en el cliente.
- Gracias a los componentes específicos para validación no es necesario transferir los datos del cliente al servidor para cada comprobación, efectuándose ésta en el cliente.
- Desde un formulario Web podemos acceder a toda la información de la solicitud enviada por el cliente gracias a la propiedad `Request`, recuperando datos sobre el cliente, parámetros asociados a la solicitud o información introducida en un formulario.
- Mediante la propiedad `Cookies` resulta fácil almacenar y recuperar información en el equipo de los clientes, a fin de identificarlos y personalizar la respuesta.
- La configuración de un formulario Web puede ser dinámica mediante el uso de archivos de configuración `web.config`.

Resumen

Aunque con diferencias lógicas por la naturaleza del tipo de aplicación, lo cierto es que las similitudes entre los formularios Windows y Web son notables como hemos podido ver en este capítulo. Esto hace mucho más fácil el diseño de interfaces nativas, ricas en funcionalidad, para aquellos clientes que pueden usar la aplicación en un PC con Windows, así como interfaces ligeras para el resto de usuarios, indistintamente del dispositivo y la plataforma hardware que utilice.

Los servicios de ASP.NET darían para escribir un libro completo, de hecho existen títulos monográficos sobre el tema. En este capítulo nos hemos ocupado de una pequeña porción: los formularios Web. Posteriormente veremos cómo usar ASP.NET para crear servicios Web, ocupándonos también, aunque de manera breve, del desarrollo de componentes para este tipo de aplicaciones.



12 Servicios de entrada y salida

Las aplicaciones no sólo cuentan con interfaces de usuario, como las abordadas en los dos capítulos precedentes, sino que deben procesar una información que, por regla general, es preciso almacenar de manera no volátil en algún dispositivo, habitualmente un disco magnético que puede encontrarse en el propio equipo del cliente o bien en el servidor al que está accediendo.

Para almacenar la información, y recuperarla cuando convenga, lo más usual es conectar con un sistema RDBMS o gestor de bases de datos enviándole las sentencias necesarias junto con los datos. Con este fin, como veremos en un capítulo posterior, podemos utilizar los servicios conocidos genéricos como ADO.NET. La alternativa, si no deseamos utilizar una base de datos, consiste en crear nosotros mismos un archivo que contenga la información.

En este capítulo conoceremos varias clases mediante las cuales podremos, en general, acceder al sistema de archivos del entorno, obteniendo información sobre las unidades lógicas existentes, sus carpetas y archivos y manipulando todos esos elementos. También aprenderemos a escribir y leer texto y datos binarios en esos archivos. Toda la metodología puede utilizarse indistintamente desde una aplicación de consola, una interfaz Windows o un componente que se ejecuta en un servidor. En los ejemplos de este capítulo, por simplicidad, recurriremos principalmente al uso de la consola como medio de entrada y salida visual.

Primera aproximación

Todas las clases de objeto, estructuras y demás elementos que necesitamos para acceder al sistema de archivos del ordenador se encuentran definidas en el ámbito `System.IO` que, por tanto, deberemos importar en cualquier proyecto donde necesitemos estos servicios.

La treintena de clases existentes pueden agruparse en tres categorías: las que facilitan información y permiten manipular las carpetas y archivos como unidad básica, aquellas que facilitan la entrada y salida de datos en los archivos y las que representan las distintas excepciones que pueden producirse. Al primer grupo pertenecen clases como `File` y `Directory` que, como puede imaginar, representan a un archivo y una carpeta. La mayoría de las clases del segundo grupo tienen un nombre que termina con el sufijo `Reader` o `Writer`, según se utilice para leer o escribir datos.

Algunos ejemplos son `BinaryReader`, `TextWriter` o `StringWriter`. Por último tenemos las clases de excepciones que, como es habitual, usan el sufijo `Exception` en sus nombres.

Como es habitual en otros servicios que ya hemos conocido, algunas de las clases de `System.IO` disponen de métodos compartidos que pueden utilizarse sin necesidad de crear objeto alguno. Estas clases, además, suelen ser `NotInheritable`, es decir, se trata de clases de uso final que no pueden utilizarse como base para derivar otras.

El paradigma de los flujos de datos

BASIC nunca ha sido un lenguaje que se haya caracterizado por sus posibilidades a la hora de trabajar con archivos en disco. La orden `Open "Archivo" For Modo As #`, conjuntamente con `Put` y `Get` o `Print` e `Input`, han estado utilizándose durante años y aún hoy muchos usuarios de Visual Basic siguen introduciéndolas en su código. Lo mismo ocurre con estructuras relativamente arcaicas, como las utilizadas para buscar archivos en carpetas o manipular esas carpetas. No es de extrañar, ya que hasta la aparición del objeto `FileSystemObject`, en Visual Basic 6, no existían muchas alternativas.

Los servicios de entrada y salida de información de la plataforma .NET, representados por las clases del ámbito `System.IO`, se basan en el paradigma del flujo o corriente de datos. Mediante métodos podemos recuperar información de ese flujo o bien introducirla en él, sin importar el medio del que estemos leyendo o donde estemos escribiendo.

Este paradigma, el del *stream* o flujo genérico, permite tratar por igual un archivo en disco, un bloque de memoria o la transferencia entre dos sistemas a través de una red. El tipo de flujo determina el medio, pero su uso no difiere de cara al programador de cualquier otro tipo de flujo.

Las posibilidades de Visual Basic .NET en este sentido, por tanto, también se han mejorado considerablemente, en cuanto a flexibilidad y facilidad, respecto a versiones previas.

Preparados para otras plataformas

Los sistemas de archivos tienen una estructura muy diferente dependiendo de la plataforma en la que trabajemos. Como usuarios de Visual Basic estamos acostumbrados a usar el sistema de archivos de Windows, en el que se utilizan letras para designar a unidades lógicas, el carácter `:` para separar el nombre de esas unidades de los caminos y el carácter \ para separar las distintas carpetas que componen un camino. Esos elementos son distintos en Linux o Mac OS, por poner un ejemplo. Es el caso del carácter separador de carpetas en los caminos. Otros elementos directamente no existen, como las letras de unidades que no tienen un equivalente en Linux o Unix.

Aunque en principio la plataforma Microsoft .NET ha aparecido para Windows, no es descartable que en un futuro también esté disponible para otros sistemas operativos. Por eso ciertos aspectos, como es el caso de las clases del ámbito `System.IO`, están desde un principio preparados para esa posible *multi-plataforma*.

Gracias a la clase `Path`, por ejemplo, es posible manipular un camino, compuesto de una secuencia de carpetas, un nombre de archivo y, posiblemente, una letra de unidad, con independencia de la plataforma, evitando problemas causados por los distintos separadores que se usan en cada plataforma.

Recuperando información del sistema de archivos

Comencemos nuestro recorrido por algunos de los servicios existentes en `System.IO` viendo cómo podemos recuperar información relativa al sistema de archivos sobre el que vamos a trabajar. Gran parte del trabajo puede efectuarse usando métodos compartidos de clases como `Directory` y `File`. Se caracterizan por ser clases de uso final y con todos los métodos compartidos, por lo que no tiene sentido crear objetos a partir de ellas. También cuentan con métodos que facilitan la manipulación de ciertos elementos, como las carpetas, aunque de esto nos ocuparemos posteriormente.

La mayoría de los métodos de esas dos clases precisan parámetros como el camino que desea explorarse o el nombre del archivo que quiere eliminarse o crearse. Si vamos a actuar repetidamente sobre un mismo elemento, una misma carpeta o un mismo archivo, generalmente podemos recurrir a las clases `DirectoryInfo` y `FileInfo`. Éstas, derivadas de `FileSystemInfo`, facilitan prácticamente las mismas operaciones e información pero mediante métodos y propiedades no compartidos, es decir, podríamos crear un objeto de estas clases para operar siempre sobre el mismo objeto.

Unidades existentes en el sistema

Suponiendo que el código que escribamos vaya a ejecutarse, ya sea en un cliente o en un servidor, en un sistema funcionando con Windows, éste contará

con una o más unidades de almacenamiento. Podemos saber qué unidades tenemos disponibles usando el método `GetLogicalDrives()` de la clase `Directory`. Esta devuelve como parámetro un arreglo de tipo `String`, por lo que podemos usar las construcciones habituales para saber el número de elementos y acceder a cada uno de ellos.

Los valores contenidos en ese arreglo tienen el formato `C:\`, es decir, la letra de unidad seguida de dos puntos y el habitual separador de camino de Windows. Pueden ser usados, por tanto, para acceder a la raíz de la unidad, la carpeta en la que están contenidas todas las demás carpetas y archivos. Podemos usar cada uno de los elementos de `GetLogicalDrives()` como camino para otros métodos que vamos a conocer de inmediato.

Enumeración de carpetas y archivos

Disponiendo de un camino, puede ser la raíz de una unidad, el camino actual recuperado con el método estático `GetCurrentDirectory()` o un camino bien conocido por nosotros, podemos obtener una lista de las carpetas y archivos que existen en él. Con ese fin se usan los métodos `GetDirectories()` y `GetFiles()` que, como `GetLogicalDrives()`, retornan un arreglo de cadenas con los nombres de las carpetas o los archivos, respectivamente.

Cada elemento de los arreglos no contiene sólo el nombre de la carpeta o el archivo, sino que éste se precederá del camino completo, incluyendo el nombre de la unidad si es aplicable. Usando sólo los cuatro métodos mencionados, todos compartidos y de la clase `Directory`, podemos crear una aplicación de consola con el código siguiente para obtener el resultado de la figura 12.1.

```
Dim Unidad, Elemento As String


Console.WriteLine("Lista de unidades en el sistema")
Console.WriteLine("--------------------------------")
For Each Unidad In Directory.GetLogicalDrives()
  Console.WriteLine(Unidad)
Next


Console.WriteLine("--------------------------------")
Console.Write("Camino actual: ")
Console.WriteLine(Directory.GetCurrentDirectory())



Console.WriteLine("--------------------------------")
Console.WriteLine("Lista de carpetas en el camino " & Unidad)
Console.WriteLine("--------------------------------")
For Each Elemento In Directory.GetDirectories(Unidad)
  Console.WriteLine(Elemento)
Next


Console.WriteLine("--------------------------------")
```

```
Console.WriteLine("Lista de archivos en el camino " & Unidad)
Console.WriteLine("------------------------------")
For Each Elemento In Directory.GetFiles(Unidad)
  Console.WriteLine(Elemento)
Next
```

```
F:\DatosTrabajo\Libros\Actuales\PBVisualBasicNET\Ejemplos\...
Lista de unidades en el sistema
-------------------------------
A:\
C:\
D:\
E:\
F:\
G:\
-------------------------------
Camino actual: F:\DatosTrabajo\Libros\Actuales\PBVisualBasicNET\Ejemplos\Cap_12\
Unidades\bin
-------------------------------
Lista de carpetas en el camino G:\
-------------------------------
G:\Archivos de programa
G:\Documents and Settings
G:\qa
G:\Inetpub
G:\PROGRAMP
G:\Proof
G:\RECYCLER
G:\System Volume Information
G:\Temp
G:\WINDOWS
-------------------------------
Lista de archivos en el camino G:\
-------------------------------
G:\bannermsdn.gif
G:\corrección.doc
G:\CURRICULUM VITAE CON FOTO.doc
G:\errorlog.log
G:\Felicitacion.jpg
G:\navidad.jpg
G:\odbcconf.log
Press any key to continue
```

Figura 12.1. Enumeración de unidades, carpetas y archivos

Si no nos interesa separar las carpetas de los archivos, en lugar de los métodos `GetFiles()` y `GetDirectories()` podemos recurrir a `GetFileSystemEntries()`. Funciona como los dos anteriores, con la diferencia de que el arreglo devuelto contendrá todas las entradas de directorio existentes, sin distinguir entre carpetas y archivos.

Nota

Los métodos `GetDirectories()`, `GetFiles()` y `GetFileSystemEntries()` cuentan con una versión sobrecargada que acepta, además de una cadena con el camino, un segundo parámetro con una plantilla de los elementos a buscar. En éste pueden utilizarse los habituales comodines ? y * para sustituir a un carácter o parte de un nombre.

Información sobre un objeto

Disponiendo del camino y nombre de un directorio, podemos utilizar los demás métodos `GetXXX()` de la clase `Directory` para obtener información adicional. En caso de que el objeto del que deseamos obtener información sea un archivo, en lugar de una carpeta, utilizaremos los métodos equivalentes de la clase `File`, también compartidos.

Cada carpeta y archivo tienen asociada una fecha de creación, una fecha de último acceso y otra de última modificación. Son datos que podemos recuperar mediante los métodos `GetCreationTime()`, `GetLastAccessTime()` y `GetLastWriteTime()`, respectivamente, de la clase `Directory` o `File` dependiendo de que sea una carpeta o un archivo. El valor es devuelto como una estructura `DateTime`.

Los archivos tienen unos atributos que determinan si puede o no escribirse en ellos, si permanecen ocultos, están encriptados o comprimidos, etc. Un mismo archivo puede contar con varios de esos atributos, por ejemplo al estar comprimido, oculto y ser sólo de lectura. El método `GetAttributes()` de la clase `File` facilita un valor en el que pueden existir varias de las constantes de la enumeración `FileAttributes`. La tabla 12.1 indica el significado de algunas de esas constantes.

Tabla 12.1. Constantes que denotan atributos de un archivo

Constante	Indica que el archivo
`Normal`	No tiene ningún atributo especial, es un archivo normal
`ReadOnly`	Es sólo de lectura
`System`	Es de sistema
`Hidden`	Está oculto
`Compressed`	Está comprimido
`Encrypted`	Está encriptado
`Temporary`	Es temporal
`Directory`	Es una carpeta
`Archive`	Esta marcado para copia
`Offline`	No está disponible
`SparseFile`	No es contiguo

La mayoría de los métodos citados disponen de métodos complementarios cuyo nombre comienza por `Set` en lugar de `Get`, siendo su finalidad facilitar la modificación del dato que corresponda. Algunas de ellas son `SetCreationTime()`, `SetLastWriteTime()` y `SetAttributes()`. Además del camino correspondiente a la carpeta o el archivo sobre el que va actuarse, dichos métodos tomarán un parámetro adicional con la fecha o los atributos, según corresponda.

Unidades, caminos y nombres

Acabamos de ver que para operar sobre una carpeta o un archivo, utilizando cualquiera de los métodos conocidos en el punto anterior y otros que trataremos posteriormente, es necesario facilitar lo que se conoce como *camino*, una

cadena de caracteres compuesta de letra de unidad, una sucesión de nombres de carpetas y subcarpetas y, finalmente, el nombre de archivo. Los caracteres utilizados para separar cada uno de esos elementos, según se indicó antes, difieren de unas plataformas a otras, por lo que extraer una parte del camino, o componerlo a partir de varios datos, pueden plantear alguna dificultad.

En lugar de intentar hacer ese trabajo manualmente, escribiendo nosotros mismos el código para obtener el nombre de un archivo, un directorio o la extensión, podemos servirnos de los métodos compartidos de la clase `Path`.

Para empezar, la clase `Path` dispone de varios miembros informativos que nos permiten saber cuáles son los separadores usados en la plataforma actual: el separador de carpetas en los caminos, el separador de caminos cuando se introducen varios, el separador de volumen y un separador alternativo. También podemos saber qué caracteres no pueden utilizarse en un nombre de archivo o carpeta. En el código siguiente, correspondiente a una aplicación de consola, se muestra el valor de cada uno de esos miembros, generando el resultado de la figura 12.2.

```
Dim Datos(,) As String = _
 {{"AltDirectorySeparatorChar", Path.AltDirectorySeparatorChar}, _
  {"DirectorySeparatorChar", Path.DirectorySeparatorChar}, _
  {"PathSeparator", Path.PathSeparator}, _
  {"VolumeSeparatorChar", Path.VolumeSeparatorChar}, _
  {"InvalidPathChars", Path.InvalidPathChars}}

Dim Indice As Integer

For Indice = 0 To Datos.GetUpperBound(0)

  Console.WriteLine("{0}= {1}", Datos(Indice, 0), Datos(Indice, 1))
Next
```

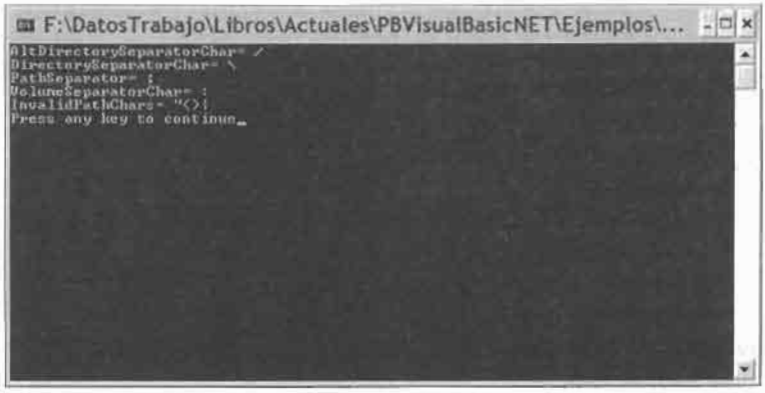


Figura 12.2. Caracteres utilizados en la composición de caminos

Aunque podríamos guiarnos por los valores de `DirectorySeparatorChar`, `VolumeSeparatorChar` y los demás miembros para recorrer un camino e intentar extraer sus elementos, lo cierto es que resulta más fácil utilizar métodos

como `GetDirectoryName()`, `GetFileName()` o `GetExtension()`. El método `HasExtension()` indica si el camino incluye o no una extensión. También podemos obtener el nombre del archivo sin extensión con el método `GetFileNameWithoutExtension()`, así como cambiar la extensión por otra con `ChangeExtension()`. Todos estos métodos toman como parámetro el camino sobre el que vaya a trabajarse.

A pesar de que poco tienen que ver con la manipulación de caminos y sus elementos, otros dos métodos interesantes de la clase `Path` son `GetTempPath()` y `GetTempFileName()`. El primero devuelve el camino que podemos usar para alojar archivos temporales, mientras que el segundo crea un nombre para un archivo temporal que no entre en conflicto con ninguno otro existente. Puede comprobar el funcionamiento de todos estos métodos como en el caso anterior, simplemente enviando los resultados a la consola para poder comprobarlos. En este caso el código podría ser el siguiente:

```
' Obtengamos el camino de la carpeta actual
Dim Carpeta As String = Directory.GetCurrentDirectory
' y recuperamos el primero de los archivos
Dim Archivo As String = Directory.GetFiles(Carpeta)(0)

' Mostramos los resultados [illegible]
Console.WriteLine( _
 "Camino completo= " & Archivo & vbCrLf & _
 "Carpeta= " & Path.GetDirectoryName(Archivo) & vbCrLf & _
 "Archivo= " & Path.GetFileName(Archivo) & vbCrLf & _
 "Archivo sin extensión= " & _
 Path.GetFileNameWithoutExtension(Archivo) & vbCrLf & _
 "Extensión= " & Path.GetExtension(Archivo))

' [illegible]
Console.WriteLine("Camino tras cambiar la extensión= " & _
 Path.ChangeExtension(Archivo, ".txt"))

' Obtengamos el camino para archivos temporales y un
' nombre de archivo temporal
Console.WriteLine( _
 "Carpeta temporal= " & Path.GetTempPath() & vbCrLf & _
 "Archivo temporal= " & Path.GetTempFileName())
```

Figura 12.3. Diseccionamos un camino en varios componentes

Manipulación de carpetas y archivos

Las clases `Directory` y `File` no sólo cuentan con miembros informativos, sino que también facilitan las operaciones que son más habituales con carpetas y con archivos: creación, cambio de nombre, eliminación, etc. Todos estos métodos necesitan como parámetro una cadena con el camino completo que identifica a la carpeta o el archivo sobre el que va a actuarse, dato que podemos haber obtenido previamente con los métodos que ya conocemos.

En el caso de la clase `Directory` los métodos que nos interesan son `CreateDirectory()`, `Move()`, `Delete()` y `Exists()`. El primero crea la carpeta o carpetas necesarias para completar el camino indicado, el segundo mueve la carpeta y todo su contenido a un nuevo punto del sistema de archivos determinado por el segundo parámetro, el tercero elimina la carpeta y, por último, el cuarto comprueba si una cierta carpeta existe o no.

La clase `File` dispone de métodos equivalentes a los que acaban de mencionarse: `Create()`, `Move()`, `Delete()` y `Exists()`, así como un método adicional, llamado `Copy()`, que nos permite copiar un archivo a otro punto o con otro nombre. También existen en `File` métodos que facilitan la apertura de un archivo para acceder a su contenido, como `Open()`, `OpenWrite()`, `OpenRead()` y `OpenText()`. Se caracterizan por retornar un objeto de la clase `FileStream` o equivalente, que conoceremos en un momento.

Cuando va a trabajarse reiteradamente en una misma carpeta o sobre un mismo archivo, en lugar de utilizar las clases `Directory` y `File` nos resultará mucho más cómodo crear un objeto de la clase `DirectoryInfo` o `FileInfo`, según corresponda. El constructor de dichas clases toma como parámetro el camino de la carpeta o archivo que va a asociarse al objeto a crear, de tal manera que posteriormente puede actuarse sobre ese elemento sin necesidad de métodos compartidos que precisan siempre el camino completo. Creado un objeto de la clase `FileInfo` asociado a un archivo, por ejemplo, podemos usar propiedades como `Attributes`, `Extension` o `LasWriteTime` para acceder a los atributos, recuperar la extensión o la fecha de última modificación. Para copiar el archivo, moverlo a otro punto o eliminarlo se utilizan métodos como `CopyTo()`, `MoveTo()` y `Delete()` que, a diferencia de los mencionados antes, no necesitan que se les indique el camino y nombre del archivo.

Acceso al contenido de los archivos

Ya sabemos cómo obtener información y manipular los archivos como unidad indisoluble, afectando a todo su contenido. Aunque en algún momento puede interesarnos cambiar de nombre, eliminar o copiar un archivo, lo cierto es que la mayoría de las aplicaciones se limitan a abrirlos, leer o escribir información de ellos y cerrarlos.

Para ello utilizaremos distintas clases de las que vamos a ocuparnos en el resto de este capítulo.

Para acceder al contenido de un archivo trabajaremos con objetos de alguna clase derivada de `Stream`. Ésta, definida en `System.IO` como el resto de las que hemos conocido hasta ahora, es una clase abstracta, es decir, no pueden crearse objetos a partir de ella, utilizada como base de las clases `FileStream`, `MemoryStream`, `BufferedStream`, `CryptoStream` y `NetworkStream`, cada una de ellas especializada en almacenar o recuperar el flujo de información de un cierto medio, como por ejemplo puede ser el disco, la memoria o bien una conexión de red.

Que todas estas clases tengan una ascendente común facilita nuestra labor, ya que conociendo los miembros de `Stream` sabremos cómo trabajar con cualquiera de sus derivadas. Es lo primero que vamos a hacer, conocer algunos de los miembros de `Stream`.

Metodología general

Como acaba de decirse, nunca crearemos un objeto de la clase `Stream` sino que nos serviremos de sus clases derivadas. Al crear un objeto de una de esas clases, como puede ser `FileStream` o `NetworkStream`, debemos facilitar al constructor un parámetro que le permita acceder al dispositivo o fuente de la que se obtendrá la información. En el caso de `FileStream`, por ejemplo, ese parámetro será un camino completo con el nombre del archivo, mientras que con `NetworkStream` el parámetro necesario es un número de *socket* TCP/IP.

Nota

Las clases `NetworkStream` y `CryptoStream`, aunque son derivadas de `Stream`, no se encuentran en el ámbito `System.IO` sino en ámbitos específicos relacionados con el trabajo en red y la seguridad.

Una vez tenemos el objeto creado, antes de realizar ninguna operación de lectura, escritura o desplazamiento podemos comprobar si dichas operaciones son posibles en el flujo que hemos obtenido. Con este fin consultaremos las propiedades `CanRead`, `CanWrite` y `CanSeek` que, como puede suponer, son de tipo `Boolean`. Las tres son también sólo de lectura.

Para leer y escribir datos en el flujo tenemos dos parejas de métodos: `ReadByte()` y `Read()`, para lectura, y `WriteByte()` y `Write()`, para escritura. Los primeros leen o escriben un solo byte, mientras que los segundos leen o escriben una secuencia de bytes. `WriteByte()` necesita como único parámetro el byte a escribir, mientras que `ReadByte()` no toma parámetros y devuelve el dato leído como un `Integer`.

Los métodos `Read()` y `Write()` toman ambos tres parámetros: un arreglo de tipo `Byte` en el que va a almacenarse o donde se encuentra la secuencia de bytes, un entero indicando el elemento de ese arreglo a partir del que se almacenará y, en tercer lugar, otro entero especificando el número de bytes a escribir o leer. En los dos casos la posición, entregada como segundo parámetro,

hace referencia al punto del arreglo donde van a alojarse los bytes leídos o se encuentran los bytes a escribir, no a la posición relativa en el flujo.

Cuando se abre un flujo de datos se crea un puntero o marcador que, generalmente, se sitúa al inicio de ese flujo. Cada vez que se efectúa una operación de lectura o escritura el marcador se desplaza hacia delante, indicando la posición donde se efectuará la próxima operación. En caso de que la propiedad `CanSeek` tenga el valor `True` podemos mover ese marcador libremente, determinando nosotros mismos cuál será el punto del flujo en el que se escriba o del que se lea.

La propiedad `Length` contiene el número de bytes de información existentes en el flujo. Conociendo este dato, podemos usar la propiedad `Position` tanto para saber cuál es la posición actual como para modificarla. Una asignación a esta última propiedad puede causar una excepción `NotSupportedException` en caso de que el flujo no permita cambiar la posición, por ello es importante comprobar antes el valor de `CanSeek`.

Además de los métodos citados para lectura y escritura, en la clase `Stream` también existen otros que facilitan el lanzamiento de operaciones asíncronas de lectura y escritura. Éstas no bloquean la ejecución del proceso principal sino que invocan a un método de retorno, ajustado al delegado `AsyncCallback`, cuando la lectura o escritura ha terminado.

Finalizado el trabajo con el flujo de datos, una llamada al método `Close()` lo cerrará liberando los recursos que pudieran haberse asignado. Este cierre, no obstante, no destruye el objeto `Stream` que estemos usando.

Apertura de archivos

Para leer y almacenar información en un archivo en disco usaremos habitualmente la clase `FileStream`. Su constructor precisa de dos a seis parámetros, dependiendo de la versión que usemos ya que se facilitan varias implementaciones distintas. Usualmente indicaremos el nombre del archivo a abrir, incluyendo el camino completo, uno de los valores de la enumeración `FileMode` indicando el modo de apertura y uno de los valores de `FileAccess` comunicando el uso que vamos a hacer del archivo. Los modos de apertura disponibles son los mostrados en la tabla 12.2, mientras que la tabla 12.3 corresponde a los valores de la enumeración `FileAccess`.

Tabla 12.2. Valores de la enumeración `FileMode` indicando el modo de apertura del archivo

Constante	Cómo se abre el archivo
`Open`	Asumiendo que existe
`CreateNew`	Creándolo o generando una excepción si existe
`Create`	*Creándolo de nuevo incluso si existe*
`OpenOrCreate`	Normal si existe o creándolo en caso contrario

Constante	Cómo se abre el archivo
Truncate	Asumiendo que existe y truncando su contenido tras la apertura
Append	Asumiendo que existe y situando el marcador al final para añadir datos

Tabla 12.3. Valores de la enumeración `FileAccess` indicando las operaciones en el archivo

Constante	Operación que podrá efectuarse en el archivo
Read	Lectura
Write	Escritura
ReadWrite	Lectura y escritura

Estos mismos valores serían los que utilizaríamos si en lugar de crear directamente un objeto de la clase `FileStream`, usando algunos de sus constructores, lo obtuviésemos mediante el método `Open()` de las clases `File` o `FileInfo`. En este último caso, con `FileInfo.Open()`, no tendríamos que facilitar el nombre del archivo y el camino ya que dicha información se facilita al crear objeto `FileInfo`.

Sabiendo cómo abrir un archivo y conociendo, aunque tan sólo de manera teórica, los métodos con que cuenta la clase `Stream`, estamos en disposición de escribir un pequeño ejemplo que tome un archivo, recupere su contenido y lo muestre por la consola. El código de este programa podría ser el siguiente:

```
Sub Main()

  Dim CaminoActual As String = Directory.GetCurrentDirectory

  Dim UnArchivo As String = Directory.GetFiles(CaminoActual)(0)
  Dim Dato As Integer, Indice As Integer


  Dim FSArchivo As FileStream = New FileStream( _
       UnArchivo, FileMode.Open, FileAccess.Read)

  Console.WriteLine("Información del archivo {0}", UnArchivo)


  For Indice = 1 To FSArchivo.Length
    Dato = FSArchivo.ReadByte

    Console.Write("{0} ({1}){2}", Dato, _
        IIf(Dato > 31, Chr(Dato), "?"), vbTab)
  Next
  Console.ReadLine()
End Sub
```

En la figura 12.4 puede ver el resultado, en este caso bastante legible porque el archivo leído no era muy extenso.

Figura 12.4. Información recuperada por el programa de un archivo

De la misma forma podríamos escribir datos en un archivo, por ejemplo un fichero temporal. La información podríamos solicitarla por la consola, como se hace en el ejemplo siguiente:

```
Sub Main()

  Dim NombreArchivo As String = Path.GetTempFileName

  Dim FSArchivo As New FileStream(NombreArchivo, _
                  FileMode.Create, FileAccess.Write)
  Dim Linea As String, Indice As Integer


  Console.WriteLine("Introduzca líneas de datos. " & _
                    "Deje la línea vacía para terminar")
  Do
    Console.Write("Dato: ")
    Linea = Console.ReadLine

    For Indice = 0 To Linea.Length - 1

      FSArchivo.WriteByte(Asc(Linea.Chars(Indice)))
    Next
  Loop While Linea <> ""

  FSArchivo.Close()


  Console.WriteLine("Información escrita en " & NombreArchivo)
End Sub
```

Observe que para escribir en el flujo de datos, el archivo, la cadena de caracteres obtenida de la consola la recorremos carácter a carácter, algo no muy eficiente. El resultado, no obstante, es el esperado ya que al tener que esperar la entrada de datos por parte del usuario el rendimiento no es precisamente un problema. En la figura 12.5 puede ver la ejecución del programa, mientras que la figura 12.6 muestra en el Bloc de notas el contenido del archivo. Fíjese en

que no hay separación entre los datos ya que los hemos escrito byte a byte, sin introducir ninguna separación.

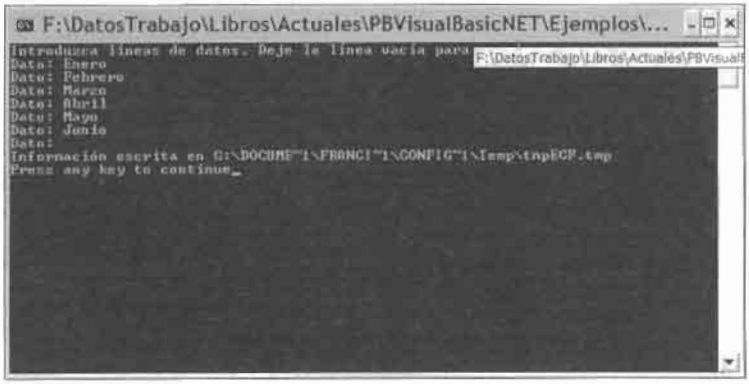


Figura 12.5. El programa escribe los datos y nos indica el archivo donde se encuentran

Figura 12.6. Contenido del archivo finalizada la ejecución del programa

Datos con tipo

Los métodos `Write()` y `Read()`, así como `WriteByte()` y `ReadWrite()`, heredados de la clase `Stream` son de bajo nivel, ya que tan sólo nos permiten trabajar con bytes individuales o secuencias de bytes. Como ha visto en el punto anterior, para escribir una cadena de caracteres es necesario tratarlos de manera individual convirtiéndolos al tipo `Byte`, o bien convirtiendo previamente la secuencia de caracteres en un arreglo de tipo `Byte`.

En el ámbito `System.IO` nos encontramos clases más especializadas, como `StreamReader` y `StreamWriter`, que facilitan la lectura y escritura de datos que se ajustan a los tipos que ya conocemos y estamos habituados a usar en variables y propiedades. *Tanto `StreamReader` como `StreamWriter` disponen* de múltiples constructores, de tal forma que podemos crear un objeto entre-

gando el camino y nombre de un archivo, un objeto derivado de `Stream` sobre el que deseamos trabajar y otros parámetros adicionales como la codificación o el tamaño del espacio intermedio de trabajo.

También podemos utilizar los métodos `OpenText()`, `CreateText()` y `AppendText()` de la clase `FileInfo`, obteniendo a cambio un objeto `StreamReader`, en el primer caso, o `StreamWriter`, en el segundo y tercero, asociado al archivo representado por el objeto `FileInfo`. También podemos utilizar los métodos homónimos de la clase `File`, en este caso facilitando como primer parámetro el camino y nombre del archivo a abrir.

Para escribir datos, utilizando un `StreamWriter`, usaremos los métodos `Write()` y `WriteLine()` que, de manera análoga a los métodos del mismo nombre de la clase `Console`, pueden tomar parámetros de distintos tipos: caracteres, números enteros y decimales, cadenas, etc. El método `WriteLine()`, además, introduce automáticamente un salto de línea al final de los datos. Con estos dos métodos, por tanto, podemos introducir cualquier información en el flujo de datos sin necesidad de efectuar conversiones previas.

La clase `StreamWriter` facilita varios métodos de lectura: `Read()`, con dos versiones distintas, `ReadLine()` y `ReadToEnd()`. El primero lee un solo carácter o un número máximo de caracteres, dependiendo de la versión que utilicemos. El segundo lee una línea completa y la devuelve como un `String`, mientras que la última lee hasta el final del flujo y lo devuelve como una cadena de caracteres. Si deseamos saber cuál es el siguiente carácter que hay en el flujo, leyéndolo pero sin hacer avanzar el marcador de posición, podemos utilizar el método `Peek()`.

Nota

Tanto `StreamWriter` como `StreamRead` utilizan un derivado de `Stream` para acceder a los datos, ya se lo hayamos facilitado nosotros mismos como parámetro o lo hayan creado ellas mismas. Podemos recuperar la referencia a ese `Stream` mediante la propiedad `BaseStream`, lo cual nos abre las puertas a usar cualquiera de los métodos que hemos conocido previamente para tratar flujos de datos.

Un visor de archivos de texto

Sirviéndonos de las dos clases que acabamos de conocer, `StreamReader` y `StreamWriter`, vamos a crear un ejemplo algo más elaborado. En este caso iniciaremos una aplicación basada en formularios Windows, siendo su finalidad permitir la lectura y modificación de archivos de texto. Para ello vamos a comenzar insertando en el formulario los componentes que pueden verse en la figura 12.7: un `ComboBox` con el título `"Seleccione una unidad"` sobre él, dos `ListBox` con los títulos `"Seleccione una carpeta"` y `"Seleccione un archivo"` sobre ellos, un `TextBox` con el título `"Contenido del archivo"` y un `Button`. Asignamos un nombre a cada uno de estos componentes

modificando su propiedad `Name`, editando también la propiedad `Anchor` para que el formulario pueda cambiarse de tamaño sin problemas y los controles aprovechen todo el espacio disponible.

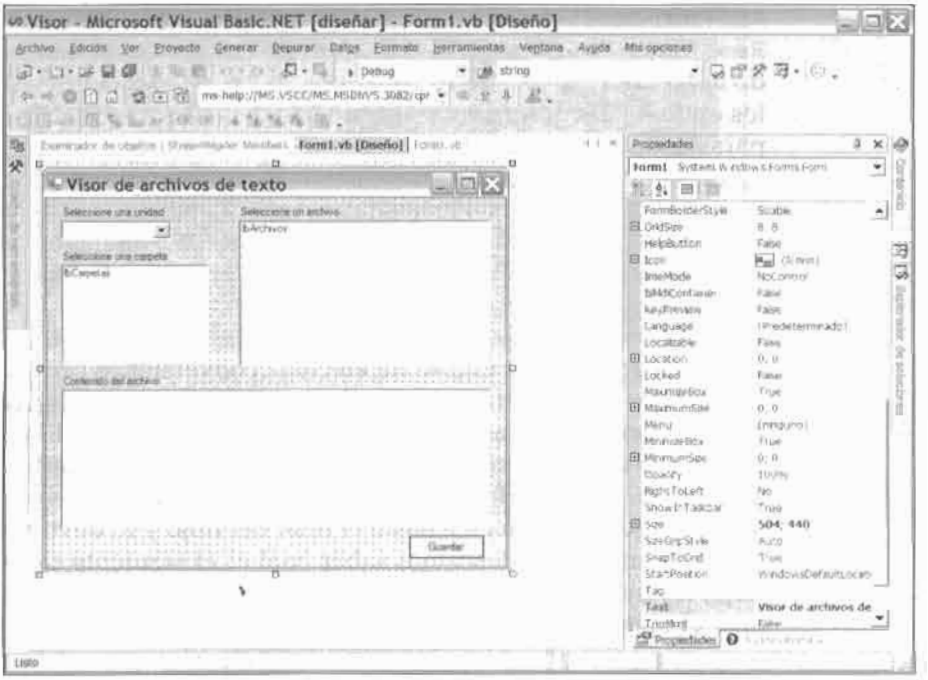


Figura 12.7. Aspecto del formulario durante la fase de diseño

Para que esta interfaz sea funcional tendremos que asignar código a distintos eventos. Éstos irán encadenándose en una secuencia que causará la actualización completa de todos los componentes que hemos dispuesto en el formulario. Para empezar, el evento de apertura del formulario lo aprovecharemos para recuperar la lista de unidades lógicas existentes en el sistema, añadiéndolas como elementos al control `ComboBox` que aparece en la esquina superior izquierda. Una selección de esa lista causará un evento, un cambio de unidad, que aprovecharemos para leer la lista de carpetas y mostrarla en la lista que hay debajo.

De manera análoga, la selección de un elemento de la lista de carpetas provocará un evento al que responderemos con la recuperación de la lista de archivos que hay en ella, mostrando los nombres en la lista que hay a la derecha. Al elegir uno de los archivos se producirá un nuevo evento que, en este caso, nos servirá para abrir dicho archivo y mostrar su contenido en el `TextBox` que hay en la parte inferior. Por último, la pulsación del botón **Guardar** tomará el contenido de la caja de texto y lo escribirá en el mismo archivo.

El código completo del programa, con comentarios explicativos, es el mostrado a continuación. Al ejecutar el programa verá aparecer de inmediato una unidad, las carpetas que hay en ella y una lista de archivos con extensión `txt`,

así como el contenido del primero de ellos. Si modifica el texto y pulsa el botón verá que los cambios se escriben en el archivo.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load
  Dim Unidad As String

  For Each Unidad In Directory.GetLogicalDrives
    cbUnidades.Items.Add(Unidad)
    If Unidad = "C:\" Then

      cbUnidades.SelectedIndex = cbUnidades.Items.Count - 1
    End If
  Next
End Sub


Private Sub cbUnidades_SelectedIndexChanged( _
   ByVal sender As System.Object, ByVal e As System.EventArgs) _
   Handles cbUnidades.SelectedIndexChanged
  Dim Carpeta As String

  lbCarpetas.Items.Clear()

  For Each Carpeta In Directory.GetDirectories(cbUnidades.Text)

    lbCarpetas.Items.Add(Path.GetFileName(Carpeta))
  Next
  lbCarpetas.SelectedIndex = 0
End Sub


Private Sub lbCarpetas_SelectedIndexChanged( _
   ByVal sender As System.Object, ByVal e As System.EventArgs) _
   Handles lbCarpetas.SelectedIndexChanged
  Dim Archivo As String
  lbArchivos.Items.Clear()

  For Each Archivo In Directory.GetFiles( _
            cbUnidades.Text & lbCarpetas.Text, "*.txt")

    lbArchivos.Items.Add(Path.GetFileName(Archivo))
  Next
  If lbArchivos.Items.Count <> 0 Then
    lbArchivos.SelectedIndex = 0
  Else
    tbContenido.Text = ""
  End If
End Sub


Private Sub lbArchivos_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles lbArchivos.SelectedIndexChanged
```

```
    ' Componemos el nombre del archivo
    Dim NombreArchivo As String = _
        cbUnidades.Text & lbCarpetas.Text & _
        Path.DirectorySeparatorChar & _
        lbArchivos.Text
    ' y lo abrimos para lectura
    Dim SRArchivo As StreamReader = _
        File.OpenText(NombreArchivo)
    ' lo leemos completo asignando el contenido al TextBox
    tbContenido.Text = SRArchivo.ReadToEnd
    SRArchivo.Close()  ' y finalmente lo cerramos
  End Sub

  ' Al pulsar el boton Guardar
  Private Sub Button1_Click(ByVal sender As System.Object, _
          ByVal e As System.EventArgs) _
          Handles Button1.Click
    ' Componemos el nombre del archivo
    Dim NombreArchivo As String = _
        cbUnidades.Text & lbCarpetas.Text & _
        Path.DirectorySeparatorChar & _
        lbArchivos.Text
    ' y lo creamos nuevo
    Dim SRArchivo As StreamWriter = _
        File.CreateText(NombreArchivo)
    ' escribimos el contenido en el archivo
    SRArchivo.Write(tbContenido.Text)
    SRArchivo.Close()  ' y finalmente lo cerramos
  End Sub
End Class
```

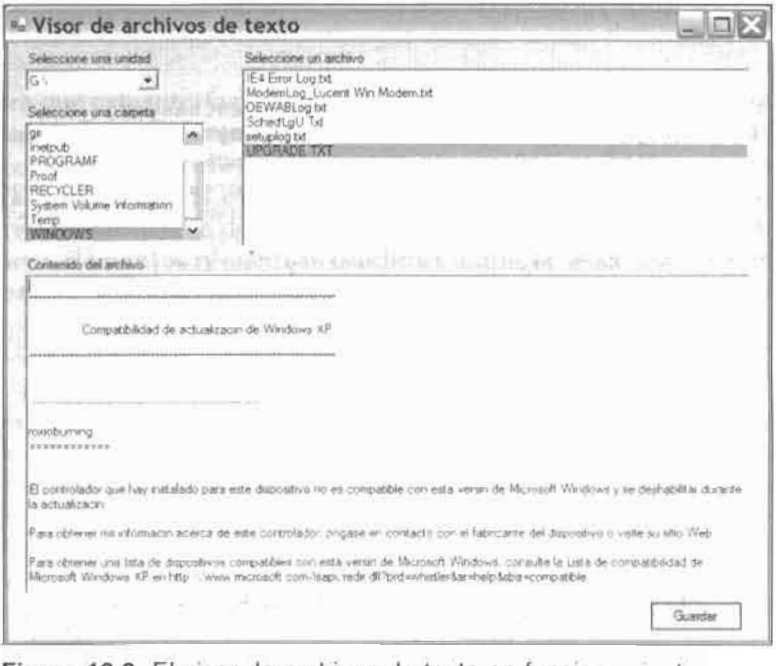


Figura 12.8. El visor de archivos de texto en funcionamiento

Flujos binarios

Las clases `StreamReader` y `StreamWriter` se caracterizan por trabajar con flujos de datos en los que la información que se lee y escribe está compuesta de secuencias de caracteres. Si usa un `StreamWriter` para, por ejemplo, escribir números y valores `Bolean`, al examinar el archivo podrá comprobar que los datos son legibles. Esto es así porque `StreamWriter` introduce en el flujo la representación textual de los datos, y no los datos en sí tal y como se almacenan en la memoria del ordenador.

En el ámbito `System.IO` existen otras dos clases, `BinaryReader` y `BinaryWriter` que, manteniendo el mismo funcionamiento básico descrito antes para `StreamReader` y `StreamWriter`, están especializados en la lectura y escritura de datos pero manteniendo su formato binario. `BinaryWriter` dispone del método `Write()` con multitud de versiones sobrecargadas aceptando distintos tipos de parámetros, como `StreamWriter`, pero la diferencia está en el formato con que se escriben los datos. En `BinaryWriter` no existe el método `WriteLine()`, ya que no está pensada para tratar con archivos de texto.

Podemos ver rápidamente la diferencia entre usar `StreamWriter` y `BinaryWriter` con un sencillo ejemplo, una aplicación de consola que ejecutaría el código siguiente:

```
Sub Main()

  Dim ArchTexto As String = Path.GetTempFileName
  Dim ArchBinario As String = Path.GetTempFileName

  Dim Datos() As Object = {25996290, 166.386, True, "Hola"}
  Dim Dato As Object

  Dim SWTexto As New StreamWriter(ArchTexto)

  Dim BWBinario As New BinaryWriter( _
     New FileStream(ArchBinario, FileMode.Create))

  For Each Dato In Datos

    SWTexto.Write(Dato)

    BWBinario.Write(Dato)
  Next

  SWTexto.Close()
  BWBinario.Close()

  Console.WriteLine("Texto -> {0}{1}Binario -> {2}", _
      ArchTexto, vbCrLf, ArchBinario)
End Sub
```

El resultado, en la consola, sería la comunicación de dos nombres de archivo. Al abrirlo encontramos el resultado de las figuras 12.9 y 12.10. La primera

corresponde al archivo generado por el `StreamWriter`, en la que podemos ver que los datos son perfectamente legibles. En la segunda, por el contrario, aparece una información no legible por nosotros.

Figura 12.9. Archivo generado por el objeto `StreamWriter`

Figura 12.10. Archivo generado por el objeto `BinaryWriter`

Complementariamente a `BinaryWriter`, la clase `BinaryReader` dispone de métodos para leer prácticamente cualquier tipo de dato: `ReadByte()`, `ReadChar()`, `ReadDecimal()`, `ReadInt32()`, `ReadString()`, `ReadBoolean()`, etc. Cada uno de ellos recupera del flujo un número fijo de bytes dependiendo del tipo de dato y efectúan la conversión adecuadamente para devolverlo.

Flujos en memoria

Ya sabe que `FileStream`, la clase que hemos usado para recuperar y almacenar información en archivos en disco, es tan sólo una de las derivadas de la

clase genérica `Stream`. Otra de las derivadas es `MemoryStream`, un flujo que utiliza la memoria como dispositivo de almacenamiento. Puede ser usada para almacenar información temporal, en lugar de usar un archivo que puede necesitar más recursos y siempre será más lento.

Como derivado de `Stream`, un objeto de la clase `MemoryStream` dispone prácticamente de los mismos miembros que hemos usados con `FileStream`. Por tanto sabemos cómo escribir y leer información, obtener la longitud, fijar la posición del marcador o cerrar el flujo. Una vez creemos un objeto `MemoryStream` éste contará con una capacidad sólo limitada por la memoria disponible en el sistema.

Dicha capacidad va ajustándose a medida que se introducen datos y podemos conocerla leyendo la propiedad `Capacity`. También es posible alterar el valor de dicha propiedad para establecer manualmente la capacidad.

En caso necesario la información contenida en un `MemoryStream` puede ser escrita en otro flujo de datos, por ejemplo un `FileStream`, almacenándola en un dispositivo no volátil.

Con ese fin se utiliza el método `WriteTo()` que, como único parámetro, necesita un objeto de una clase derivada de `Stream`.

Otra alternativa, para trabajar con flujos de datos en memoria, consiste en utilizar las clases `StringWriter` y `StringReader`. Éstas son similares a `StreamWriter` y `StreamReader`, contando prácticamente con los mismos miembros para lectura y escritura, pero en lugar de usar un `Stream` como medio de almacenamiento se sirven de un `StringBuilder` que, al fin y al cabo, no es más que una secuencia de caracteres en memoria.

Puntos clave

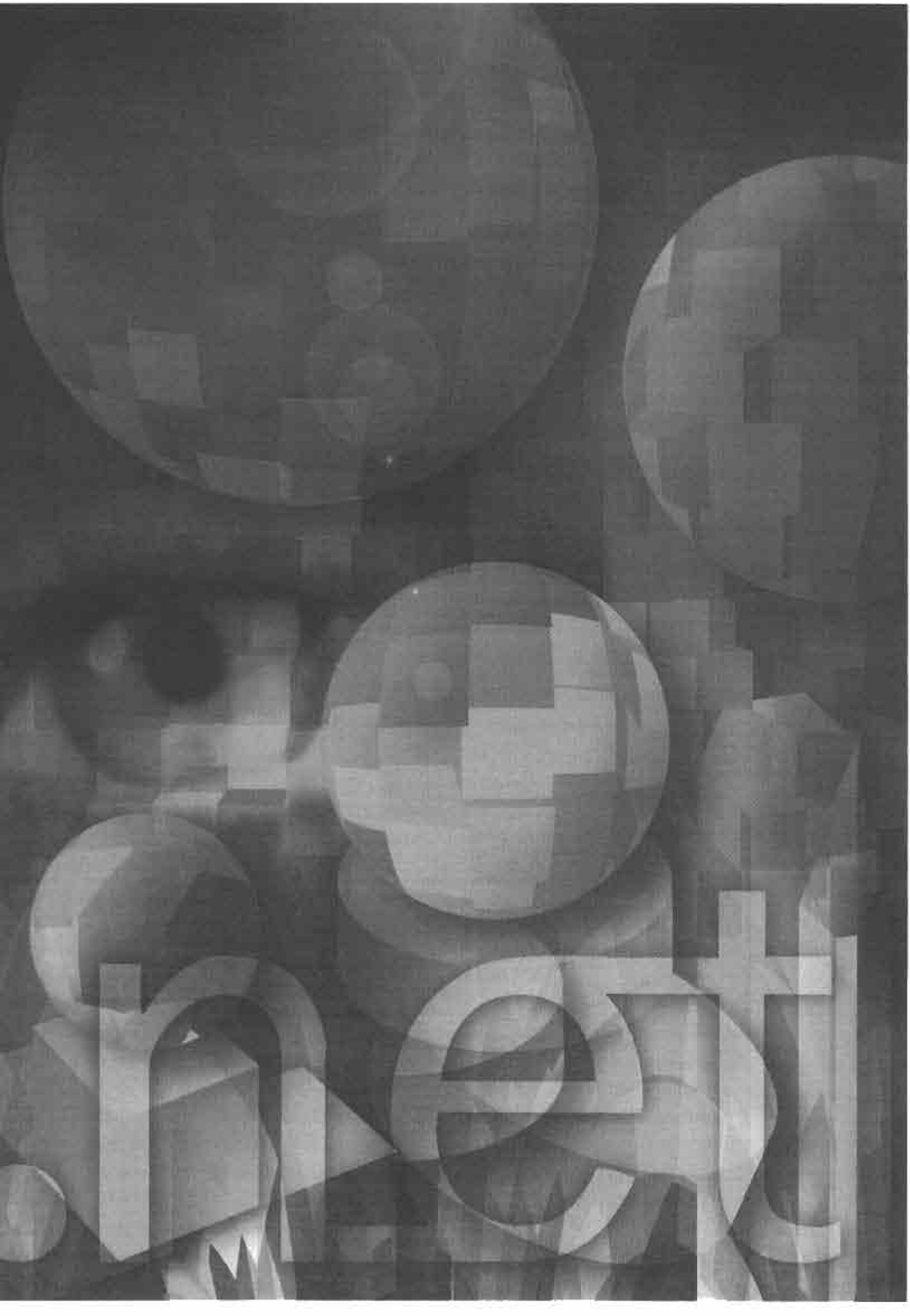
- El ámbito `System.IO` de la plataforma .NET aloja una serie de clases que facilitan la manipulación de carpetas y archivos, así como el uso de flujos de datos para gestionar la entrada y salida de información.
- Las clases `Directory` y `File` están compuestas exclusivamente de métodos estáticos que facilitan información sobre carpetas y archivos, permitiendo también las operaciones de manipulación básicas: creación, borrado y renombrado.
- Mediante las clases `DirectoryInfo` y `FileInfo` es posible crear objetos asociados a carpetas o archivos, simplificando el trabajo reiterado con estos elementos sin necesidad de tener que facilitar el camino y nombre en cada paso.
- Todas las clases que gestionan flujos de datos en diversos dispositivos o medios están derivadas de `Stream`, por lo que cuentan con un grupo de miembros comunes.
- Para trabajar con archivos en disco se utiliza la clase `FileStream`, mientras que para mantener información en memoria existe `MemoryStream`.

- Existen varias clases, derivadas de `TextReader` y `TextWriter`, que facilitan las operaciones de lectura y escritura en los flujos de datos. Algunas de estas clases son `StreamReader`, `StreamWriter`, `BinaryReader`, `BinaryWriter`, `StringReader` y `StringWriter`.

Resumen

Este capítulo nos ha servido para conocer los elementos más importantes del ámbito `System.IO`, imprescindibles para poder almacenar y recuperar información de archivos en disco. En general, usaremos estas clases cuando necesitemos manipular el sistema de archivos o bien trabajar con información no alojada en bases de datos.

Como veremos en un capítulo posterior, Visual Studio .NET dispone de todos los elementos necesarios para poder trabajar con bases de datos como Access o SQL Server, siendo el mecanismo preferente a la hora de mantener de forma permanente los datos de la aplicación. Los elementos conocidos en este capítulo, por tanto, tienen un uso más secundario.



13
Elaboración de gráficos

Si en el capítulo anterior apuntábamos que los servicios de entrada y salida de Visual Basic .NET suponían una mejora considerable, respecto a los existentes en versiones previas, lo mismo podríamos decir al tratar los servicios para elaboración de gráficos. Hasta la versión 6.0 prácticamente seguíamos contando con las mismas posibilidades gráficas que existían en los tiempos del GW-BASIC, con órdenes como `Point`, `Line` o `Circle` y poco más. Las versiones de BASIC existentes en muchos microordenadores de hace veinte años tenían más posibilidades gráficas que Visual Basic 6, una herramienta presentada a finales de los noventa.

Como consuelo, paliando en parte la falta de servicios gráficos en el propio lenguaje, quedaba la posibilidad de usar las funciones del GDI (*Graphics Device Interface*), nombre con el que se denomina genéricamente al conjunto de servicios de Windows relacionados con gráficos. El inconveniente es que para ello se precisa la definición previa, en Visual Basic, de centenas de funciones, constantes, enumeraciones y tipos de datos, ya que no existe ninguna biblioteca de tipos ni recurso equivalente que facilite su uso desde este entorno.

Las posibilidades gráficas de Visual Basic .NET son las de la propia plataforma .NET, muy superiores a las de versiones previas y a las de GDI. Estos servicios se agrupan en una serie de ámbitos con nombres a los que genéricamente se llama GDI+. Basta una importación de esos ámbitos en nuestro proyecto para tener a nuestro alcance todo lo necesario para la elaboración de gráficos en el plano bidimensional, la manipulación de imágenes, impresión, etc. Será el tema que nos ocupe este capítulo, dedicado a la visualización de esos gráficos, y el próximo, centrado en los temas de impresión.

Ámbitos relacionados con GDI+

Los servicios de GDI+ forman parte de la plataforma Microsoft .NET, lo cual significa que están disponibles para cualquier lenguaje de programación .NET y, asimismo, desde cualquier tipo de proyecto.

Esto implica que podemos utilizar GDI+ tanto para dibujar en un formulario Windows como para enviar un gráfico en una página web, indistintamente con Visual Basic .NET, C#, Visual C++ .NET, J# o bien cualquier otro lenguaje .NET.

Estos servicios se alojan en su mayor parte en un ensamblado que se aloja en el módulo `System.Drawing.Dll`, ensamblado que consta de varios ámbitos con nombre que estructura las clases de objetos y tipos según la finalidad concreta para la que están diseñados.

El ámbito de primer nivel, del cual *cuelgan* los demás relacionados con GDI+, es `System.Drawing`. Utilizando el Examinador de objetos de Visual Basic .NET (véase figura 13.1) puede acceder al contenido de esos ámbitos para conocer sus miembros.

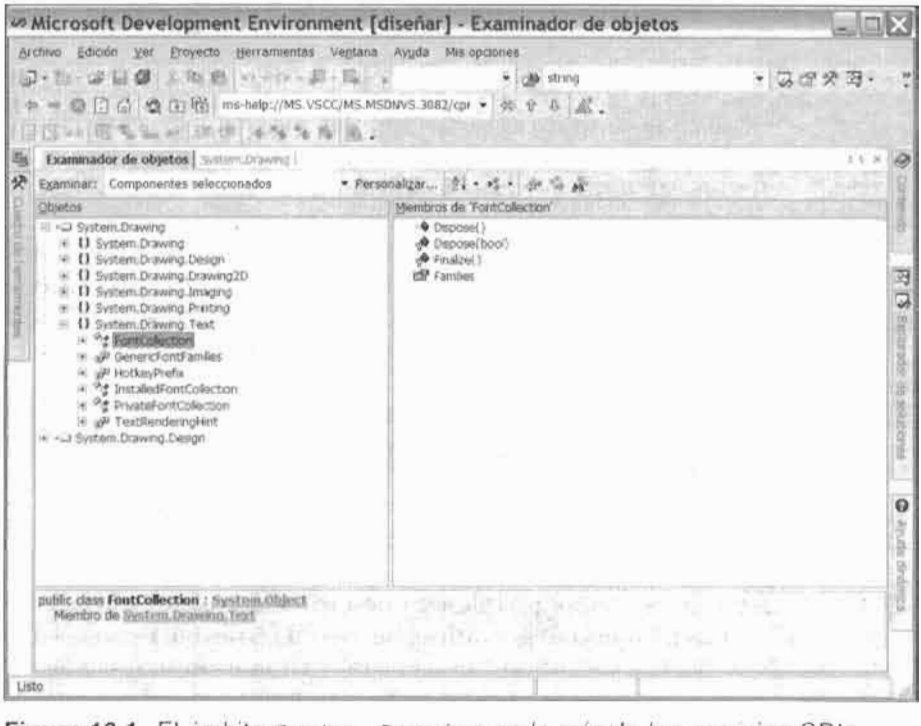


Figura 13.1. El ámbito `System.Drawing` es la raíz de los servicios GDI+

Para comenzar, vemos brevemente cuál es el contenido de cada uno de los ámbitos existentes en `System.Drawing` para adquirir una visión general de lo que tenemos a nuestra disposición.

Elementos de propósito general

Elementos de propósito general son los que encontramos en el propio ámbito `System.Drawing`, en forma de clases, enumeraciones y estructuras que resultan básicas a la hora de trabajar con gráficos, independientemente de su naturaleza.

Entre las estructuras más usadas están `Point` y `Rectangle`. La primera de ellas se utiliza para mantener y manipular las coordenadas de un punto, mientras que con la segunda también se opera sobre unas dimensiones. `Size` es complementaria de `Point`, almacenando unas dimensiones sin un punto de origen, como sí tiene `Rectangle`. Estas tres estructuras almacenan internamente las coordenadas y dimensiones como un `Integer`. Existen tres equivalentes, llamadas `PointF`, `RectangleF` y `SizeF` que, manteniendo la misma finalidad y funcionalidad, usan un `Single` para almacenar los datos.

`Color` es otra de las estructuras básicas. Con sus miembros es posible componer cualquier color, ya sea utilizando algunas de las constantes predefinidas con nombres de colores o a partir de sus componentes de rojo, verde y azul.

Las clases `Pen`, `Brush`, `Font` y `Bitmap` representan entidades básicas de dibujo, en este caso el lápiz para pintar bordes y contornos, la brocha para los rellenos, las características del texto o los puntos de una imagen. En los miembros `Pens` y `Brushes` tenemos un conjunto predefinido de lápices y brochas.

Otra de las clases importantes es `Region`, mediante la cual puede representarse cualquier figura gráfica compuesta de otras entidades. Las regiones facilitan el escalado y transformación de los gráficos.

Todos los elementos anteriores se utilizan para dibujar sobre una superficie, un concepto que, en el caso de GDI+, se representa mediante un objeto de la clase `Graphics`. Esta cuenta con multitud de métodos que facilitan el dibujo de entidades relativamente simples: líneas rectas, arcos, curvas, polígonos y cadenas de caracteres. Después veremos cómo obtener un objeto `Graphics` para dibujar sobre un formulario Windows.

Nota

También es posible dibujar directamente sobre una imagen en memoria, lo que se conoce habitualmente como un *bitmap* o mapa de bits.

Gráficos vectoriales y 2D

Cuando los elementos existentes en `System.Drawing` no son suficientes para poder trazar gráficos más complejos, basados en caminos, regiones, brochas complejas, fundidos de color y, en general, gráficos en dos dimensiones de cierta complejidad, tendremos que recurrir entonces a los elementos del ámbito `System.Drawing.Drawing2D`.

Con las clases `HatchBrush`, `PathGradientBrush` y `LinearGradientBrush` podemos crear brochas para rellenos complejos, utilizando patrones y

degradados de color. Estos pueden ser lineales, de un color a otro siguiendo una línea recta horizontal, vertical o inclinada, o bien seguir un cierto camino, por ejemplo con una figura no regular.

Usando las clases `AdjustableArrowCap` y `CustomLineCap` podemos definir elementos gráficos a medida para dibujar en los extremos de las líneas.

Otra de las clases que tenemos en ese ámbito es `Matrix`. Con ella es posible definir matrices sobre las que efectuar posteriormente traslaciones, rotaciones y escalados. Estas operaciones pueden aplicarse a entidades gráficas, consiguiendo efectos de cambio de tamaño, rotación en cualquier eje y modificación de posición en el espacio bidimensional.

Más adelante, en este mismo capítulo, veremos cómo usar algunos de estos elementos para crear efectos gráficos que en Visual Basic 6 requerirían un trabajo mucho mayor.

Manipulación de imágenes

Tras el ámbito anterior, siguiendo el orden en que aparecen en el **Examinador de objetos**, nos encontramos con otro llamado `System.Drawing.Imaging`. En su interior hay clases para el tratamiento de imágenes tanto vectoriales como basadas en mapas de bits.

Gracias a un mecanismo basado en codificadores y descodificadores totalmente extensible, representado por la clase `Encoder`, es virtualmente posible trabajar con cualquier formato gráfico. En principio, es posible recuperar imágenes de tipo `BMP`, `EMF`, `GIF`, `JPG`, `PNG`, `TIF` y `WMF`. Usando clases como `ColorMap` y `ColorPalette` se definen las paletas de colores, mientras que `BitmapData` contiene los datos de la imagen propiamente dicha e `ImageAttributes` sus atributos.

Usando la clase `Metafile` podemos crear archivos de gráfico vectorial registrando en ellos todas las primitivas para su composición. Este registro puede ser, posteriormente, reproducido sobre una superficie de dibujo.

También encontramos en este mismo ámbito las clases necesarias para tratar con paletas de color, indispensables cuando los puntos que componen una imagen no representan valores de color real sino que actúan como índices de una tabla limitada de colores.

Impresión

El ámbito `System.Drawing.Printing`, del que nos ocuparemos con detalle en el próximo capítulo tal y como se indicaba antes, contiene los elementos necesarios para hacer posible la publicación de información.

Con las clases existentes en dicho ámbito podremos configurar los parámetros de impresión, preparar una vista previa del documento y, finalmente, enviarlo a un dispositivo de impresión que, en la práctica, no tiene por qué ser una impresora, ya que existen dispositivos de impresión que actúan como intermediarios para el envío de documentos por fax, la generación de documentos digitales tipo PDF, etc.

Tipografías

El último de los ámbitos relacionado con GDI+ es `System.Drawing.Text` que, como puede suponer, está relacionado con operaciones con cadenas de caracteres y tipografía en general. El texto puede ser tratado como una entidad gráfica más, siendo posible su uso en un lienzo visible, como puede ser un formulario; en una imagen oculta o en una superficie de impresión.

En `System.Drawing.Text` tan sólo encontramos las clases necesarias para determinar los tipos de letra que hay en el sistema y sus atributos, ya que los métodos para trabajar con texto forman parte del grupo de elementos básicos descrito con anterioridad, en el `System.Drawing`.

Primera aproximación

Saber qué ámbitos están relacionados con GDI+ y más o menos cuál es su contenido nos permite tener una noción general de a dónde debemos recurrir en cada caso. Veamos ahora, en la práctica, cómo usar algunos de los elementos mencionados utilizando el lenguaje Visual Basic .NET. Por simplicidad en principio dibujaremos directamente sobre el área cliente de un formulario Windows aunque, como veremos después, existen otras posibles superficies. Los puntos siguientes describen, paso a paso, el proceso para dibujar algunas entidades simples sobre un formulario vacío. Con ello aprenderemos a preparar algunos elementos básicos, como el pincel o la brocha, usándolos sobre una superficie de dibujo.

Cuando va a utilizarse como superficie de dibujo el formulario de una aplicación, sin usar ningún componente específico, el lugar adecuado para insertar el código de dibujo suele ser un método asociado al evento `Paint` del formulario.

Inicie una nueva aplicación Windows, acceda al módulo de código asociado al formulario, pulsando el correspondiente botón del Explorador de soluciones y, a continuación, despliegue la lista que hay en la parte superior izquierda del editor eligiendo el elemento (Eventos de clase base). Al hacerlo, en la lista de la derecha aparecerán todos los eventos de la clase base de nuestro formulario, es decir, de la clase `Form`. Seleccionando el evento `Paint` generaremos un método con el siguiente encabezado:

```
Private Sub Form1_Paint(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.PaintEventArgs) _
  Handles MyBase.Paint
```

Tal y como se indica al final de la cabecera, con la palabra clave `Handles`, este método será ejecutado cuando se produzca el evento `Paint` de la clase base, representada por `MyBase`. Esto significa que el código que incluyamos aquí se ejecutará cada vez que sea necesario redibujar el interior del formulario.

El segundo parámetro, una variable de la clase `PaintEventArgs`, es el que más nos interesa en este caso. Su propiedad `Graphics` nos facilita la superficie

de dibujo, un objeto de la clase `Graphics` mencionada anteriormente. En la propiedad `ClipRectangle`, una estructura `Rectangle`, se indica el área de dibujo a la que debemos ceñirnos.

Usando la propiedad `Graphics` del parámetro `e`, por tanto, tenemos acceso a la superficie de dibujo y podríamos usar los métodos de esa clase para dibujar cualquier entidad básica.

Preparación de un pincel

Para dibujar prácticamente cualquier entidad, incluso la más simple, antes necesitaremos crear un pincel o lápiz, definiendo las propiedades que tendrá. Sería la acción equivalente, si dibujásemos a mano, a buscar y coger el lápiz adecuado para lo que queremos pintar. En GDI+ un lápiz es un objeto de la clase `Pen`, objeto que crearíamos como cualquier otro.

Los distintos constructores de que dispone la clase `Pen` permiten crear el lápiz especificando su color, el grueso o incluso una brocha. Gracias a la tecnología *Intellisense* es fácil saber qué parámetros necesitamos e, incluso, seleccionarlos directamente de una lista. En la figura 13.2 puede ver cómo una etiqueta emergente nos indica que debemos facilitar un color para el lápiz, mientras que una lista emergente, aparecida al introducir la palabra `Color` seguida de un punto, nos ofrece una enumeración de los colores disponibles.

Figura 13.2. Composición de la sentencia para crear un lápiz

Suponiendo que deseemos crear un pincel de color azul celeste y de cinco puntos de grosor, podríamos usar la sentencia siguiente para obtenerlo en la variable `Pincel`.

```
Dim Pincel As Pen = New Pen(Color.Aqua, 5)
```

Dibujo de entidades simples

Disponiendo ya del lápiz, podríamos usar algunos de los métodos de la clase `Graphics` para dibujar en la superficie, en este caso el formulario, entidades sencillas como líneas, rectángulos o arcos. El nombre de todos estos métodos comienza con el prefijo `Draw`, por lo que le será fácil encontrarlos en la lista de

métodos (véase figura 13.3). Al introducir el nombre de cualquier método y abrir paréntesis obtendrá una ayuda sobre los parámetros que debe facilitar.

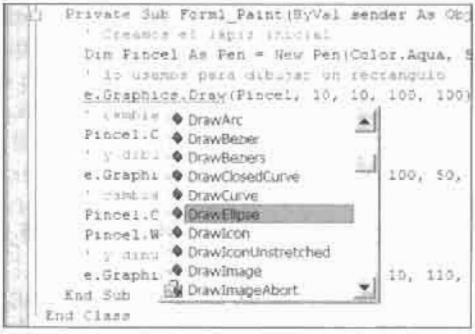


Figura 13.3. Lista de métodos de dibujo de la clase `Graphics`

Suponiendo que hemos dado los pasos anteriores, encontrándonos en la ventana de código con el método correspondiente al evento `Paint` abierto, podríamos utilizar el código siguiente para trazar algunas entidades sencillas y obtener el resultado que puede verse en la figura 13.4. Observe cómo se utiliza la misma variable `Pincel` para modificar las características del objeto y dibujar cada entidad en un color diferente.

```
Private Sub Form1_Paint(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.PaintEventArgs) _
  Handles MyBase.Paint

  Dim Pincel As Pen = New Pen(Color.Aqua, 5)

  e.Graphics.DrawRectangle(Pincel,10,10,100,100)

  Pincel.Color = Color.DarkCyan
  ' y dibujamos un arco
  e.Graphics.DrawArc(Pincel,100,100,50,50,0,180)

  Pincel.Color = Color.Blue
  Pincel.Width = 8

  e.Graphics.DrawLine(Pincel,10,10,110,110)
End Sub
```

Relleno de áreas

Los métodos que tienen el prefijo `Draw` siempre trazan contornos, usando para ello los atributos del lápiz que se les entrega como primer parámetro. Algunos de ellos, como `DrawRectangle()`, `DrawPolygon()` o `DrawClosedCurve()` cuentan con métodos complementarios en los que se cambia el prefijo `Draw` por `Fill`. Algunos de dichos métodos son `FillRectangle()`, `FillPolygon()` y `FillClosedCurve()`.

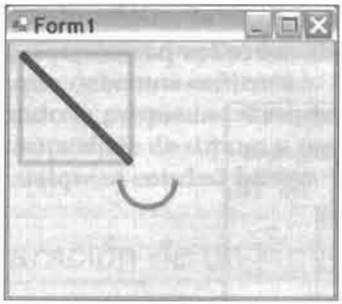


Figura 13.4. Aspecto del formulario con las tres entidades gráficas

Estos métodos no dibujan un contorno, sino que rellenan la figura especificada usando una cierta brocha. La brocha es un objeto, de la clase `Brush`, que se entrega como primer parámetro a estos métodos.

Existen una serie de brochas predefinidas accesibles mediante las propiedades de la clase `Brushes`, todas ellas de sólo lectura. Se caracterizan por ser brochas sólidas de colores estándar, de tal manera que podemos usarlas sin necesidad de crear un objeto de la clase `Brush` y establecer sus propiedades.

Nota

En realidad la clase `Brush` es abstracta y, por tanto, no podemos crear directamente objetos con ella. Existen, no obstante, una serie de clases derivadas mediante las cuales, como veremos luego, es posible crear brochas más complejas.

Usando el código del ejemplo anterior como base, ponga delante de la llamada al método `DrawRectangle()` la sentencia siguiente:

```
e.Graphics.FillRectangle(Brushes.Gold, 10, 10, 100, 100)
```

Lo que hacemos es rellenar el mismo rectángulo de color oro, trazando después su contorno, de color azul celeste, con el método `DrawRectangle()`. Al ejecutar el programa verá la diferencia.

Añadimos un texto

Aunque en capítulos previos hemos visto cómo incluir títulos, un texto cualquiera, en un formulario utilizando el control `Label`, lo cierto es que ésta no siempre será la mejor opción, especialmente si es mucho texto y con diferentes atributos. Hay que tener en cuenta que al usar un control `Label` se asignan una serie de recursos intrínsecos por su naturaleza de componentes visual, recursos que no son necesarios para insertar un texto en la superficie de dibujo del formulario usando el método `DrawString()`.

Antes de poder escribir el texto, no obstante, tendremos que crear un objeto de la clase `Font` estableciendo los atributos que se utilizarán a la hora de escribirlo. Al igual que ocurre con otros objetos, la clase `Font` dispone de múltiples constructores. Utilizando uno de ellos podemos indicar el nombre del tipo de letra, su tamaño en puntos y el estilo: negrita, cursiva, subrayada, etc.

Los parámetros para insertar el texto, llamando al método `DrawString()`, serán cinco: la cadena de texto a escribir, el objeto `Font` con los atributos, un objeto `Brush` con la brocha para pintar el texto y dos enteros indicando las coordenadas donde se comenzará a dibujar.

Sirviéndonos una vez más del ejemplo anterior como base, añada las sentencias siguientes al final y ejecútelo de nuevo. El aspecto será similar al de la figura 13.5. En este caso el texto tiene un color uniforme porque así es la brocha seleccionada pero, en la práctica, el texto podría haberse dibujado con un cierto patrón, degradado de color o incluso mapa de bits.

```
Dim TipoLetra As Font = New Font("Courier New", 14, _
  FontStyle.Underline Or FontStyle.Bold)

e.Graphics.DrawString("GDI+ y Visual Basic.NET", _
                      TipoLetra, Brushes.Blue, 10, 200)
```

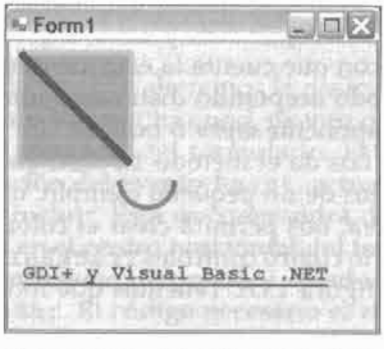


Figura 13.5. Aspecto del formulario con el texto

En estos cuatro puntos ha visto cómo crear un gráfico en un formulario con gran simplicidad, si bien hemos conocido tan sólo unos elementos mínimos. Vamos a centrarnos ahora en el estudio de las características de algunos de estos elementos.

Elementos básicos

Ahora que sabemos básicamente cómo dibujar sobre la superficie de un formulario, y podemos usar este conocimiento para ir componiendo nuestros ejemplos, vamos a analizar con mayor detalle los elementos indispensables que deberemos utilizar a la hora de elaborar gráficos. Algunos de ellos, como por

ejemplo el lápiz y el tipo de letra, los hemos conocido brevemente en los puntos previos.

Forman parte de este grupo de elementos básicos todos los atributos que pueden afectar a las entidades gráficas en el momento de dibujarlas: coordenadas, color, tipo de trazo del contorno y trama de relleno o estilo tipográfico al escribir un texto. Como en el ejemplo anterior, seguiremos utilizando el evento `Paint` del formulario y los métodos de la clase `Graphics` para dibujar.

Colores

Todas las entidades gráficas, desde una línea hasta un texto, se dibujan en la superficie utilizando un cierto color o combinación de colores. Esta información, además, es fundamental para la mayoría de las composiciones gráficas. Los colores se representan en GDI+ mediante la estructura `Color`, definida en el ámbito `System.Drawing`. El color es almacenado internamente, en dicha estructura, como tres valores separados: intensidad de verde, rojo y azul. También existe un componente, conocido como *alpha*, el cual determina el nivel de transparencia. Los cuatro componentes son de tipo `Byte`, es decir, estarán comprendidos entre 0 y 255. Esto nos permite trabajar con algo más de dieciséis millones de colores, lo que se conoce habitualmente como *color real*.

A la hora de definir un color podemos utilizar distintas técnicas, siendo una de ellas el uso de los componentes rojo, verde y azul. Con este fin se utiliza el método estático `FromArgb()` con que cuenta la estructura `Color`. Existen diferentes versiones de este método aceptando distintos parámetros, según nos interese facilitar además el componente *alpha* o no.

Para ver la flexibilidad que nos da el método `FromArgb()` en la composición de colores vamos a servirnos de un pequeño ejemplo, un formulario Windows que, de manera interactiva, nos permita crear el color que nos interese. Insertaremos en dicho formulario cuatro controles `TrackBar` con cuatro `Label` encima, como se aprecia en la figura 13.6. Tenemos que modificar los valores siguientes de los `TrackBar`:

- `Orientation` – Establece la orientación del control que, por defecto, es horizontal. La cambiaremos a `Vertical`.
- `Maximum` – Un control `TrackBar` representa gráficamente un valor, el indicado por su cursor, que estará comprendido entre los límites indicados por las propiedades `Minimum` y `Maximum`. La primera tiene por defecto el valor `0`, que es el que nos interesa, pero la segunda tenemos que modificarla asignándole el valor `255`.
- `TickStyle` – Por defecto estos controles muestran unas marcas que sirven como guía. En este caso no nos interesan y, por ello, asignaremos el valor `None` a esta propiedad para desactivarlas.

Finalmente asignaremos un nombre diferente a cada `TrackBar`, modificando su propiedad `Name`, y estableceremos la propiedad `Text` de los cuatro `Label` que nos sirven como títulos.

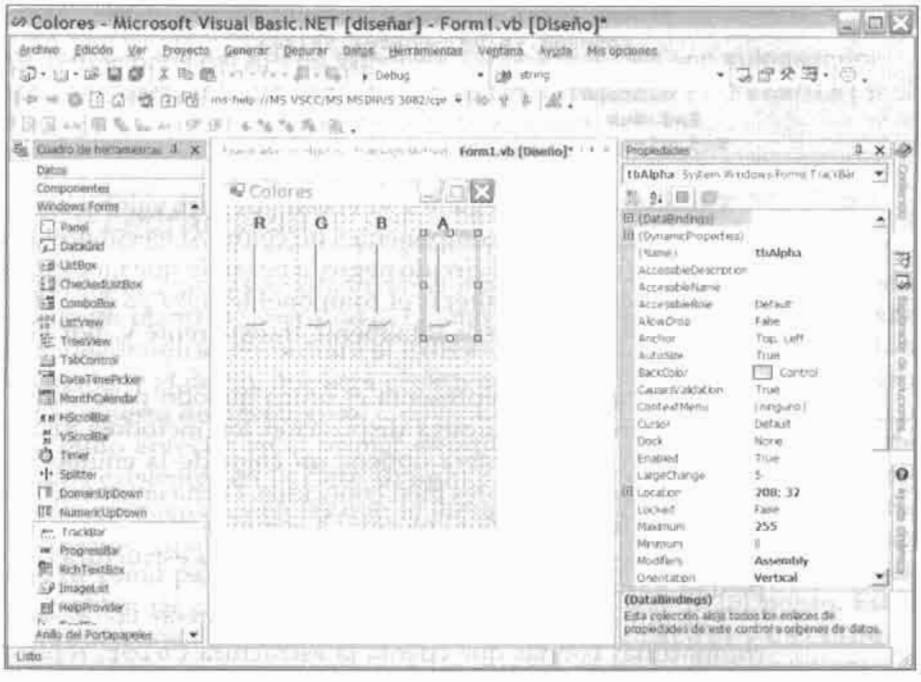


Figura 13.6. Aspecto del formulario durante la fase de diseño

Cada vez que alteremos la posición del cursor de un `TrackBar` se generará el evento `ValueChanged`, evento que aprovecharemos para llamar al método `Invalidate()` del formulario. Éste invalida el contenido actual y fuerza la generación del evento `Paint`, actualizando así la composición gráfica que vamos a incluir. Ésta se compondrá de un círculo relleno de color negro, más o menos en el centro horizontal del formulario, sobre el que dibujaremos un rectángulo relleno de un color creado a partir de los valores establecidos en los `TrackBar`. El código necesario es el siguiente:

```
Private Sub tbRojo_ValueChanged( _
   ByVal sender As Object, ByVal e As System.EventArgs) _
   Handles tbRojo.ValueChanged, tbVerde.ValueChanged, _
   tbAzul.ValueChanged, tbAlpha.ValueChanged

  Invalidate()
End Sub


Private Sub Form1_Paint(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.PaintEventArgs) _
    Handles MyBase.Paint

  Dim Brocha As New SolidBrush(Color.Black)

  e.Graphics.FillEllipse(Brocha, 60, 160, 150, 150)
```

```
        Brocha.Color = Color.FromArgb(tbAlpha.Value, tbRojo.Value, _
            tbVerde.Value, tbAzul.Value)

        e.Graphics.FillRectangle(Brocha, 60, 160, 150, 150)
    End Sub
End Class
```

Observe cómo entregamos a `FromArgb()` los valores actuales para el componente *alpha* y los tres componentes de color. Al ejecutar el programa en principio no verá más que el círculo negro, a pesar de que modifique los componentes de color. Tendrá que alterar el componente *alpha* ya que, al ser inicialmente cero, la brocha obtenida es totalmente transparente y, por ello, el rectángulo no llega a verse.

`FromArgb()` no representa el único método para obtener un cierto color, también tenemos a nuestra disposición los métodos `FromKnownColor()` y `FromName()`. La primera obtiene un color de la enumeración `KnownColor`, que pasan por ser colores bien conocidos. Dicha enumeración contiene decenas de constantes referidas a colores. La segunda toma como parámetro una cadena con el nombre de un color, retornando una estructura `Color` que en su propiedad `Name` tiene dicho nombre.

Por último tenemos la alternativa de utilizar las decenas de propiedades compartidas con las que cuenta la estructura `Color`, representando cada una de ellas un color determinado. Así, podemos usar `Color.Blue` en lugar de `Color.FromArgb(0,0,255)` para obtener el mismo color, y de forma análoga `Color.Beige`, `Color.Fuchsia` o `Color.Magenta` sin necesidad de conocer los componentes RGB de dichos colores.

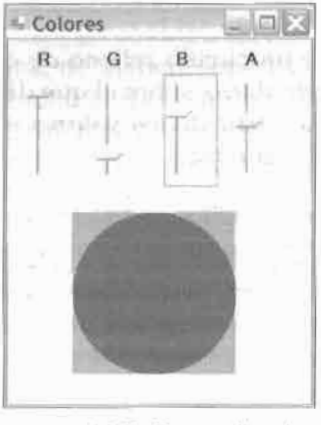


Figura 13.7. Dependiendo del componente *alpha* el rectángulo será más o menos transparente y permitirá ver el círculo que hay debajo

Nota

En el ámbito `System.Drawing` existe una clase, llamada `ColorTranslator`, que cuenta con diversos métodos compartidos que facilitan la con-

versión de colores cuando se trabaja no sólo con GDI+ sino también con HTML o funciones del API de Windows. Dichos métodos son autodescriptivos: `FromHtml()`, `ToHtml()`, `FromWin32()`, `ToWin32()`, `FromOle()` y `ToOle()`.

Puntos y coordenadas

Las entidades gráficas se sitúan, en el momento de dibujarse, en una cierta posición de un plano bidimensional. La mayoría de ellas precisan dos puntos: el de partida, donde se encontrará el extremo de la línea o una de las esquinas del rectángulo, y el de fin, que será el extremo opuesto o fin de la línea, o bien la esquina opuesta del rectángulo.

Como pudo apreciar en el primer ejemplo de este capítulo, la mayoría de los métodos existentes en la clase `Graphics` aceptan como parámetros individuales las coordenadas en las que se dibujará la entidad gráfica. Esos métodos, no obstante, cuentan con versiones sobrecargadas que aceptan también un `Rectangle` como parámetro, o dos `Point`. Como sabe, éstas son estructuras que representan las coordenadas de un área o la posición de un punto. Estas estructuras disponen de métodos como `Offset()`, `Inflate()` y `Ceiling()` que facilitan diversas operaciones con los puntos, como el desplazamiento, y las áreas, incrementando la superficie o reduciéndola.

Cada vez que dibujamos una entidad, facilitando para ello unas coordenadas, éstas sufren una serie de transformaciones. La aplicación utiliza un sistema de coordenadas conocido como *world*, personalizable en origen, orientación y unidad de medida. Este sistema de coordenadas, el del mundo de la propia aplicación, es transformado primero al sistema de coordenadas de la página imaginaria en la cual va a dibujarse y, finalmente, al sistema de coordenadas del dispositivo final, pudiendo éste ser la pantalla, la impresora u otro similar.

Por defecto el sistema de coordenadas lógicas de nuestra aplicación, el conocido como *world*, usa el punto de pantalla como unidad de medida y, además, coincide en el origen y orientación del sistema de coordenadas de una ventana. Por eso la esquina superior izquierda es el origen de coordenadas que, en sentido positivo, avanzan hacia la derecha y abajo.

Es posible establecer el origen de coordenadas en cualquier otro punto del plano, utilizando para ello el método `TranslateTransform()` de la clase `Graphics`. Éste toma como parámetro dos números decimales indicando el desplazamiento del origen en el eje X y el eje Y. El siguiente fragmento de código, por ejemplo, establece el origen de coordenadas en el centro del formulario, de tal forma que las coordenadas a la izquierda de ese centro serán negativas y las que estén a la derecha positivas, ocurriendo exactamente igual en el eje vertical. El resultado puede verlo en la figura 13.8.

```
e.Graphics.TranslateTransform( _
    Me.ClientSize.Width / 2, Me.ClientSize.Height / 2)
```

```
e.Graphics.FillRectangle(Brushes.Aqua, 0, 0, 100, 100)

e.Graphics.FillRectangle(Brushes.Black, -100, -100, 100, 100)
```

Figura 13.8. El origen de coordenadas se encuentra en el centro del área de dibujo

La unidad de medida usada para dibujar, que por defecto es el *pixel* o punto de pantalla, también es un parámetro modificable. Para seleccionar otra unidad tan sólo tenemos que modificar el valor de la propiedad `PageUnit` de `Graphics`, asignando uno de los valores de la enumeración `GraphicsUnit`. De esta forma podremos trabajar en milímetros, pulgadas, puntos de impresora, puntos de documento, etc. Usando una unidad de medida estándar, como puede ser el milímetro, nosotros indicaremos el tamaño real que deseamos que tengan las entidades.

La transformación de ese sistema de coordenadas al utilizado finalmente en el dispositivo se encargará de calcular los puntos necesarios que, por ejemplo, serían distintos en la pantalla que en la impresora puesto que el número de puntos por pulgada, o DPI, de una impresora es bastante superior al de una pantalla.

Nota

Podemos conocer el número de puntos por pulgada del objeto `Graphics` sobre el que estamos trabajando consultando las propiedades sólo de lectura `DpiX` y `DpiY`. La primera facilita la densidad de puntos horizontal y la segunda el mismo dato en vertical.

Por último mencionar la propiedad `PageScale`, que almacena un valor mediante el que se establece el factor de escalado a la hora de convertir las coordenadas de página al dispositivo final.

El valor inicial es 1, de tal manera que, en la práctica, no se aumenta ni disminuye la proporción de las entidades. Asignando un valor superior verá que las entidades crecen, mientras que cualquier valor inferior a uno reducirá sus dimensiones.

Brochas

Otro de los elementos usados habitualmente al dibujar, principalmente al rellenar superficies, es la brocha. Antes se indicaba que no era posible crear un objeto de la clase `Brush` dado que es una clase abstracta, definida a propósito para servir como base de otras y no para crear objetos directamente a partir de ella. Las clases derivadas de ésta son cinco:

- `SolidBrush` – Es la brocha sólida de un determinado color. El constructor acepta como único parámetro el color, atributo que podemos modificar mediante la propiedad `Color`.
- `HatchBrush` – Brocha en la que se combina un determinado patrón, con un color de primer plano, y un color de fondo. Las propiedades `HatchStyle`, `ForegroundColor` y `BackgroundColor` alojan estos parámetros. En la enumeración `HatchStyle` encontramos la lista de patrones posibles, en total más de medio centenar.
- `LinearGradientBrush` – Se usa para componer brochas sólidas en las que se efectúan transiciones o fundidos entre colores. El fundido es lineal entre dos o más colores. La línea puede ser vertical, horizontal o diagonal en cualquier sentido. Un total de ocho constructores distintos permiten establecer los parámetros de este tipo de brocha.
- `PathGradientBrush` – Es una brocha similar a la anterior ya que se efectúan fundidos de colores sólidos, si bien en este caso el fundido no es lineal sino que va desde el centro hacia los bordes de un cierto camino.
- `TextureBrush` – Seguramente el tipo de brocha más espectacular, ya que se utiliza una imagen como patrón repetitivo. La imagen, representada por un objeto `Image`, puede recuperarse de un archivo en disco o bien dibujarse. En cualquier caso, antes de ser utilizada como brocha pueden aplicársele transformaciones.

La mejor forma de ver cómo se usan este tipo de brochas es sirviéndonos de un ejemplo basado, una vez más, en un formulario Windows. En el método correspondiente al evento `Paint` incluimos el código mostrado a continuación.

```
Private Sub Form1_Paint(ByVal sender As Object, _
ByVal e As System.Windows.Forms.PaintEventArgs) _
Handles MyBase.Paint

    Definimos la área base para dibujar los distintos rectángulos
  Dim Rectangulo As Rectangle = New _
    Rectangle(10, 10, 100, 100)

    Una brocha de transición, entre dos colores en sentido diagonal
  Dim BrochaFundido As LinearGradientBrush = _
   New LinearGradientBrush(Rectangulo, _
   Color.Yellow, Color.Red, _
   LinearGradientMode.BackwardDiagonal)
```

```
    Dim BrochaSimple As SolidBrush = _
      New SolidBrush(Color.FromArgb(128, 0, 64))

    Dim BrochaPatron As HatchBrush = _
      New HatchBrush(HatchStyle.Plaid, _
      Color.YellowGreen)

    Dim BrochaImagen As TextureBrush = _
      New TextureBrush( _
      New Bitmap("C:\Windows\Nubes.bmp"))

    Dim Tipo As Font = _
      New Font("Courier New", 24, FontStyle.Bold)

    With e.Graphics
      .FillRectangle(BrochaFundido, Rectangulo)
      Rectangulo.Offset(100, 0)
      .FillRectangle(BrochaSimple, Rectangulo)
      Rectangulo.Offset(-100, 100)
      .FillRectangle(BrochaPatron, Rectangulo)
      Rectangulo.Offset(100, 0)
      .FillRectangle(BrochaImagen, Rectangulo)

      .DrawString("Brochas", Tipo, _
        BrochaImagen, 20, 210)
      .DrawString("Brochas", Tipo, _
        BrochaFundido, 120, 230)
    End With
  End Sub
```

La primera parte del código, previa a la sentencia `With`, se encarga de crear una variable `Rectangle` con las coordenadas y dimensiones del rectángulo a dibujar, varias brochas distintas y un tipo de letra. A continuación, en el bloque `With/End With`, utilizamos el método `FillRectangle()` para dibujar cuatro rectángulos usando las distintas brochas. Observe cómo se utiliza el método `Offset` de `Rectangle` para desplazar las coordenadas y no tener que calcularlas manualmente.

Finalmente, usando el método `DrawString()`, se dibujan dos cadenas de texto con dos brochas distintas. Como se aprecia en la figura 13.9, una brocha no sirve sólo para rellenar entidades gráficas sino que, además, pueden aportar mucha vistosidad a un simple texto.

Figura 13.9. Cuatro rectángulos rellenos con diferentes brochas

Lápices

A diferencia de lo que ocurre con las brochas, no existe una clase base y múltiples clases específicas de lápices. La única clase disponible, `Pen`, es, sin embargo, suficientemente flexible como para permitir la personalización muy detallada de este elemento básico. Aparte de diferentes colores y grosores, un lápiz puede tener trazo discontinuo, contar con elementos gráficos en los extremos tales como flechas y, además, usar brochas para el trazo, de tal manera que se abre la posibilidad de usar lápices con patrones, imágenes o fundidos de color.

Como ya sabe, el color y grosor de un lápiz se establece con las propiedades `Color` y `Width`. Si no deseamos un trazo continuo podemos modificar la propiedad `DashStyle`, consiguiendo así una línea a trazos, punteada, con una combinación de trazos y puntos o, incluso, con un estilo definido a medida. No tiene más que consultar los valores de la enumeración `DashStyle` para comprobar las posibilidades que existen. Mediante la propiedad `DashCap` del lápiz también es posible definir si el extremo de cada trazo será plano, redondeado o triangular. Mediante las propiedades `StartCap` y `EndCap` se indican los terminadores que aparecerán en los extremos de las líneas que dibujemos con este lápiz. Un terminador no es más que un elemento gráfico como puede ser una flecha, un triángulo o un bolo. En este caso los valores posibles son los de la enumeración `LineCap`.

Nota

Algunos de los elementos a los que está haciéndose referencia se encuentran en el ámbito `System.Drawing.Drawing2D`, por lo que deberá añadir la correspondiente sentencia `Imports` al inicio de su módulo de código para poder acceder a ellos.

En lugar de utilizar un color sólido para efectuar los trazos, un lápiz puede servirse de una brocha previamente definida. De esta manera los trazos pueden usar fundidos de colores, patrones de relleno o incluso imágenes.

Como en el caso anterior, para conocer en la práctica algunas de las posibilidades de los lápices vamos a servirnos de un ejemplo. Comenzaremos insertando en un formulario los elementos que pueden observarse en la figura 13.10: dos etiquetas de texto y dos controles `ComboBox`. Éstos nos servirán como listas desplegables en las que mostraremos todos los tipos de trazo y terminadores disponibles, facilitando la selección de cualquiera de ellos.

Lo primero que tenemos que hacer, cuando se abra esta ventana, será recuperar todos los elementos de las enumeraciones `DashStyle` y `LineCap` para añadirlos a las listas desplegables. Las enumeraciones son objetos derivados de la clase `System.Enum`, una clase en la que se implementan diversos métodos que, por lo tanto, heredan todas las enumeraciones. Uno de esos métodos es `GetValues()`, mediante el cual es posible obtener un arreglo con todos los valores que contiene la enumeración. Este arreglo se devuelve en forma de objeto de la clase `Array`, que es la base de todos los arreglos.

Una vez que ya tenemos el arreglo con los valores, recorrerlo no es ningún problema. Podemos, por ejemplo, usar la instrucción `For Each` y recuperarlos individualmente. Lo que tenemos, sin embargo, son valores, datos que podemos usar, llegado el caso, para efectuar asignaciones a las propiedades `DashLine`, `StartCap` o `EndCap`. Lo que necesitamos añadir a las listas desplegables, sin embargo, es una cadena de texto, no el valor de una enumeración. En cualquier momento, independientemente del tipo de valor sobre el que estemos trabajando, podemos recuperar una representación textual mediante el método `ToString()` del que disponen prácticamente todos los objetos.

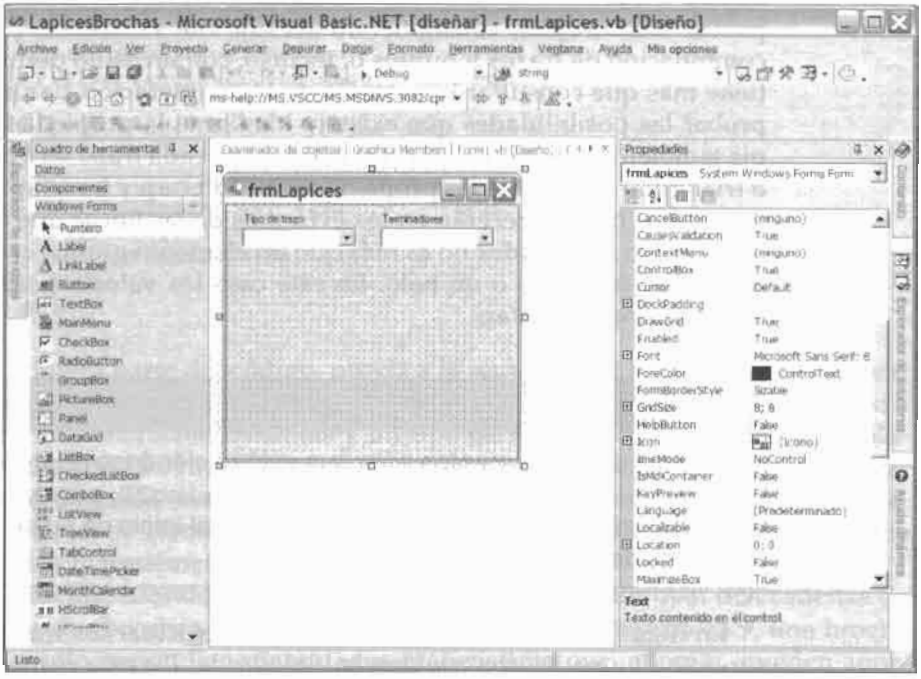


Figura 13.10. Aspecto del formulario a diseñar

Sabiendo esto, comenzaremos añadiendo a la clase del formulario los cuatro miembros siguientes. Los dos primeros, de tipo `Array`, alojarán la lista de valores de las citadas enumeraciones, mientras que `Terminador` y `Trazo` mantendrán el valor elegido en cada momento.

```
Private Terminadores As Array
Private Trazos As Array

Private Terminador As LineCap
Private Trazo As DashStyle
```

Los cuatro miembros son privados, de tal manera que sólo podrán utilizarse desde la clase en que han sido declarados, es decir, desde el interior de los métodos de este formulario.

Acto seguido hacemos doble clic sobre el formulario, o bien seleccionamos el evento `Load` usando las listas que hay en la parte superior del editor de código, e introducimos las sentencias mostradas a continuación.

```
Private Sub frmLapices_Load( _
ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load

  Terminadores = System.Enum.GetValues(GetType(LineCap))

  Trazos = System.Enum.GetValues(GetType(DashStyle))

  For Each Terminador In Terminadores

    cbTerminadores.Items.Add(Terminador.ToString())
  Next

  For Each Trazo In Trazos
    cbTrazo.Items.Add(Trazo.ToString())
  Next

  cbTerminadores.SelectedIndex = 0
  cbTrazo.SelectedIndex = 0
End Sub
```

Utilizamos el método `GetValues()` de la clase `Enum` dos veces, una para recuperar la lista de valores de la enumeración `LineCap` y otra para los de la enumeración `DashStyle`. Observe que lo que necesita dicho método como parámetro es un tipo de dato, no el identificador de la enumeración en sí. Por ello usamos la función `GetType()` para recuperarlo.

Después, en sendos bucles, añadimos a las listas desplegables los nombres de cada uno de los valores de las enumeraciones. Terminamos seleccionando

el primer elemento de cada una de las listas, utilizando para ello la propiedad `SelectedIndex`.

El tipo de trazo y terminador, almacenados en las variables `Trazo` y `Terminador` respectivamente, los usaremos en un método del formulario, asociado al evento `Paint`, para dibujar dos líneas en la ventana. Una de ellas, recta, estará basada en una brocha con un patrón de relleno, mientras que la otra utilizará un degradado de color. El código de ese método es el siguiente:

```
Private Sub frmLapices_Paint( _
ByVal sender As Object, _
ByVal e As System.Windows.Forms.PaintEventArgs) _
Handles MyBase.Paint

  Dim BrochaPatron As HatchBrush = _
    New HatchBrush(HatchStyle.Sphere, Color.Olive)

  Dim Lapiz As Pen = New Pen(BrochaPatron, 8)

  Dim Rectangulo As Rectangle = New Rectangle(250, 150, 100, 150)

  Dim BrochaFundido As LinearGradientBrush = _
   New LinearGradientBrush(Rectangulo, _
   Color.Yellow, Color.Red, _
   LinearGradientMode.BackwardDiagonal)

  Lapiz.DashStyle = Trazo
  Lapiz.StartCap = Terminador
  Lapiz.EndCap = Terminador

  e.Graphics.DrawLine(Lapiz, 50, 150, 200, 250)

  Lapiz.Brush = BrochaFundido
  e.Graphics.DrawArc(Lapiz, 250, 150, 100, 150, -90, 180)
End Sub
```

Si ejecuta el programa efectivamente verá, como en la figura 13.11, la línea recta y el arco con los atributos que hemos establecido en el método anterior. Al seleccionar uno de los elementos de las listas que hay en la parte superior, sin embargo, no vemos cambio alguno, no se modifica ni el tipo de trazo ni los elementos que deberían aparecer en los extremos.

Lo que ocurre es lógico puesto que no hemos asociado código alguno al evento que generan las listas desplegables cuando se cambia la selección actual, concretamente el evento `SelectedIndexChanged`. Seleccione en la lista de la izquierda uno de los controles `ComboBox` y en la de la derecha el mencionado evento, repitiendo la operación con el otro `ComboBox`. Los dos métodos deberían quedar así:

```
Private Sub cbTrazo_SelectedIndexChanged( _
ByVal sender As Object, _
ByVal e As System.EventArgs) _
Handles cbTrazo.SelectedIndexChanged

  Trazo = Trazos(cbTrazo.SelectedIndex)
  Invalidate()
End Sub


Private Sub cbTerminadores_SelectedIndexChanged( _
ByVal sender As Object, _
ByVal e As System.EventArgs) _
Handles cbTerminadores.SelectedIndexChanged
  Terminador = Terminadores(cbTerminadores.SelectedIndex)
  Invalidate()
End Sub
```

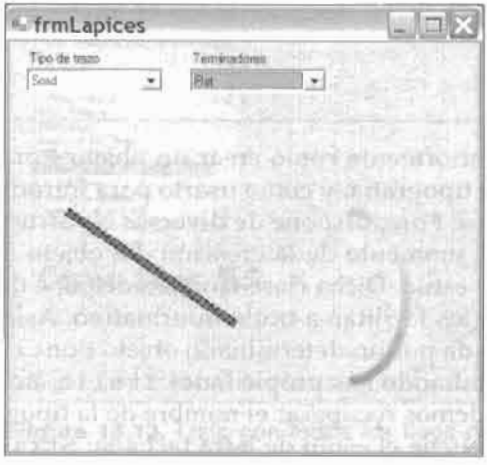


Figura 13.11. Aspecto de las líneas con los atributos por defecto

Recuperamos el valor seleccionado en cada caso y, a continuación, llamamos al método `Invalidate()` del formulario. Este fuerza la generación del evento `Paint` y, por tanto, la actualización del contenido dibujado en la ventana. Ahora sí, al seleccionar valores en las listas desplegables, podrá ver cómo las líneas cambian su trazo o terminadores. En la figura 13.12 puede ver cómo el trazo es discontinuo y en los extremos aparecen puntas de flecha.

Nota

Aunque en este ejemplo se ha utilizado un mismo tipo de terminador para ambos extremos de las líneas, como denota la existencia de las propiedades `StartCap` y `EndCap` sería posible usar dos diferentes, uno para el punto de inicio y otro para el final.

Figura 13.12. Las líneas tras modificar el tipo de trazo y los terminadores

Tipos de letra

Ya hemos visto anteriormente cómo crear un objeto `Font`, para establecer las características de la tipografía, y cómo usarlo para introducir un texto en el lienzo de dibujo. La clase `Font` dispone de diversos constructores que nos permiten especificar, en el momento de la creación del objeto, la familia del tipo de letra, su tamaño y el estilo. Dicha clase también dispone de propiedades que recuperan esos datos y los facilitan a título informativo. Así, podemos saber si la tipografía representada por un determinado objeto `Font` está en cursiva, negrita o subrayada consultando sus propiedades `Italic`, `Bold` y `Underline`. De manera análoga podemos recuperar el nombre de la tipografía: `Name`; su tamaño: `Size`, o saber si tiene el estilo de letra tachada: `StrikeOut`.

Lo único que necesitamos saber, de cara a crear un objeto `Font`, es la lista de tipografías disponible en el contexto gráfico sobre el que trabajamos. Normalmente utilizaremos un `FontDialog`, un cuadro de diálogo estándar, para permitir al usuario seleccionar no sólo la familia sino también el resto de atributos del texto. Podemos, no obstante, obtener una lista de tipos disponibles utilizando el miembro compartido `Families` de la clase `FontFamily`. Cada uno de los elementos de la colección representada por `Families` es un objeto `FontFamily` que, entre otros elementos, cuenta con una propiedad llamada `Name` que aloja el nombre del tipo.

Asumiendo que en un formulario hemos insertado un `ListBox` y un `Label`, sin modificar propiedad alguna, podríamos usar el código siguiente para mostrar en la lista todos los tipos de letra disponibles y que, al seleccionarse cualquiera de ellos, la etiqueta de texto lo utilizase. En la figura 13.13 puede ver el programa en funcionamiento.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
```

```
    Dim Tipo As FontFamily
    For Each Tipo In FontFamily.Families
      ' añadimos a la lista
      ListBox1.Items.Add(Tipo.Name)
    Next
  End Sub

  Private Sub ListBox1_SelectedIndexChanged( _
     ByVal sender As System.Object, ByVal e As System.EventArgs) _
     Handles ListBox1.SelectedIndexChanged
    ' modificamos el tipo de letra usado por el Label
    Label1.Font = New Font(ListBox1.Text, 20, FontStyle.Bold)
    ' y ponemos el nombre del tipo de letra
    Label1.Text = ListBox1.Text
  End Sub
```

Figura 13.13. Lista con todos los tipos de letra disponibles

Además de la clase `Font`, para definir la apariencia del texto, y el método `DrawString()` de la clase `Graphics`, para mostrarlo en la posición del plano que deseemos, lo único adicional que podemos necesitar será calcular el espacio necesario para un cierto texto. Para obtenerlo utilizaremos el método `MeasureString()` de `Graphics` facilitando dos parámetros: la cadena de caracteres y el objeto `Font` que usaremos para mostrarla. El resultado será una estructura `Size` con el ancho y alto necesarios.

Manipulación de mapas de bits

En un ejemplo previo hemos visto, muy brevemente, cómo recuperar una imagen de un archivo y cómo usarla para crear una brocha. La clase `Bitmap`, utilizada para abrir un archivo y leer la imagen, es una de las descendientes de `Image`, una clase genérica para el trabajo con imágenes. Otras clases derivadas

de ella son `Metafile` e `Icon` que, como puede suponer, están especializadas en el tratamiento de meta-archivos e iconos. Puesto que todas estas clases derivan de `Image`, comparten un conjunto de miembros común. Lógicamente, cada clase derivada aporta elementos específicos. La clase `Metafile`, por ejemplo, dispone de un método llamado `PlayRecord` no existente en `Bitmap` y cuya finalidad es reproducir las instrucciones de un meta-archivo.

Nota

Los conocidos como meta-archivos son archivos gráficos que en lugar de contener información sobre los colores de cada punto de una imagen, como ocurre con los formatos `BMP`, `GIF` o `JPG`, almacenan las órdenes necesarias para reproducir una determinada composición gráfica.

Recuperación de una imagen

La clase `Image` dispone de diferentes métodos para recuperar una imagen, ya sea desde un archivo, un flujo de datos o a través del manejador de un mapa de bits clásico de Windows. Estos métodos son `FromFile()`, `FromStream()` y `FromHbitmap()`, respectivamente. Estos tres métodos son estaticos, o compartidos, lo cual significa que podemos usarlos simplemente poniendo delante el nombre de la clase, sin crear previamente un objeto. Usando las clases derivadas tenemos otras alternativas. Tanto `Bitmap` como `Metafile` son clases que disponen de decenas de constructores públicos. Con ellos es posible crear el objeto recuperando la imagen de un archivo, creándola inicialmente vacía, obteniéndola a partir de un recurso incorporado en el proyecto, etc.

Si no queremos tan solo recuperar la imagen para mostrarla sino que, además, nos interesa usarla como superficie sobre la que dibujar, deberemos crear un objeto de la clase `Graphics` usando el método compartido `FromImage()`. Éste toma como único parámetro un objeto `Image`, o de una clase derivada, y retorna un objeto `Graphics`. Dicho objeto actúa como intermediario para poder trabajar sobre la imagen, de tal manera que cualquier invocación a métodos de `Graphics` se reflejará en la imagen contenida en el objeto `Image`.

La operación complementaria a la recuperación de una imagen, la salvaguarda o escritura en un archivo, se efectúa mediante el método `Save()`. Éste toma como único parámetro el nombre del archivo aunque, opcionalmente, también puede especificarse el formato de archivo que se desea obtener. También existe la posibilidad de utilizar un flujo, un derivado de `Stream`, como destino de la información.

Información y manipulación

Tras recuperar la imagen, con cualquiera de los mecanismos citados antes, podemos obtener información diversa acerca de ella sirviéndonos de algunas propiedades de la clase `Image` o la derivada que estemos utilizando.

Las dimensiones de la imagen, un dato especialmente interesante si pretendemos mostrarla en una ventana, pueden recuperarse de las propiedades `Width` y `Height`, ancho y alto respectivamente. También la propiedad `Size` contiene esta información, si bien en una estructura `Size` en lugar de como dos enteros independientes. Podemos asignar el contenido de la propiedad `Size`, por ejemplo, a la propiedad `AutoScrollMinSize` de un formulario para ajustar las barras de desplazamiento en la ventana a fin de permitir la visualización de imágenes de mayores dimensiones que las disponibles en dicha ventana.

Otras propiedades interesantes, desde el punto de vista informativo, son `VerticalResolution`, `HorizontalResolution`, `PixelFormat`, `Palette` y `Flags`. Las dos primeras nos facilitan la resolución, expresada en puntos por pulgada, que tiene esta imagen tanto vertical como horizontalmente. Mediante `PixelFormat` sabremos cuántos bits necesita cada uno de los puntos de la imagen para almacenar su información de color, conociendo el número de colores, si existe un canal con información de transparencia, etc. En caso de que la imagen utilice una paleta de color, la propiedad `Palette` permite tanto obtenerla como modificarla. Por último, en la propiedad `Flags` se almacenan indicadores de atributo adicionales.

Como se ha indicado en el punto anterior, uno de los mecanismos que podemos usar para modificar una imagen consiste en obtener un objeto `Graphics` a partir de ella. La clase `Image`, no obstante, también cuenta con algún método capaz de efectuar alteraciones globales a la imagen. El método `RotateFlip()`, por ejemplo, nos permite invertir la imagen en cualquiera de sus dos ejes, vertical u horizontal, o bien girarla en pasos de 90, 180 ó 270 grados.

Visualización de las imágenes

Tras crear un objeto `Image`, o derivado, mediante cualquiera de los métodos que ya conocemos, lo que tenemos es una imagen alojada en el interior de un objeto, pero dicho objeto no es visible. Un objeto `Image` no es un formulario ni un componente con interfaz de usuario, por lo que si deseamos poder ver la imagen, como es previsible, deberemos *volcarla* desde el objeto `Image` en algún otro que sí sea visible.

La clase `Graphics`, con la que hemos trabajado previamente, dispone de múltiples implementaciones de un método llamado `DrawImage()`. Con él podemos mostrar una imagen en un objeto `Graphics`, por ejemplo asociado a la superficie de un formulario, respetando las dimensiones originales, alterándolas para ajustarse a un cierto tamaño, situándola en una cierta posición, dibujando sólo una porción de la imagen, etc.

Veamos cómo utilizar este método, y algunos de los descritos previamente de la clase `Image`, con un sencillo ejemplo. Su finalidad será permitir la visualización de cualquier gráfico que podamos recuperar de un archivo, rotarlo en pasos de 90 grados e incluir en él un título. Comenzaremos, como en casos anteriores, iniciando un nuevo proyecto Visual Basic .NET basado en formularios Windows. El único elemento adicional que precisaremos, aparte del formulario, será un componente `OpenFileDialog`.

Ya que vamos a estar manipulando una imagen de un formato desconocido a priori, añadiremos al inicio de la clase una variable privada de tipo `Image`:

```
Private Imagen As Image
```

En principio esta variable estará vacía, por lo que poco podemos hacer con ella. Cuando se detecte un clic de ratón sobre el formulario, al generarse el evento `Click`, usaremos el `OpenFileDialog` para mostrar un cuadro de diálogo que permita seleccionar un archivo gráfico. El nombre del archivo elegido estará en la propiedad `FileName`, dato que usamos tanto para establecer el título de la ventana como para crear un objeto `Image` guardando la referencia en la variable `Imagen`.

Como se puede ver en el código de este método, mostrado a continuación, recuperamos las dimensiones de la imagen y las asignamos a la propiedad `AutoScrollMinSize` del formulario. De este modo aseguramos la existencia de unas barras de desplazamiento que faciliten el acceso a toda la imagen en caso de que ésta tenga unas dimensiones superiores a las de la ventana. Finalmente, usamos el método `Invalidate()` para forzar la actualización del contenido del formulario.

```
Private Sub Form1_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles MyBase.Click

  If OpenFileDialog1.ShowDialog() = DialogResult.OK Then

    Me.Text = OpenFileDialog1.FileName

    Imagen = Image.FromFile(OpenFileDialog1.FileName)

    Me.AutoScrollMinSize = Imagen.Size

    Invalidate()
  End If
End Sub
```

La imagen, alojada en el objeto `Image`, no será visible por el simple hecho de que la variable `Imagen` pertenezca al formulario. Necesitamos, por tanto, dibujarla en el interior del formulario y, para ello, utilizaremos el mismo sistema descrito en puntos previos: el evento `Paint`. Antes de llamar al método `DrawImage()` facilitando como parámetro la variable `Imagen`, no obstante, hay que tener en cuenta que, en principio, ésta no contiene un dato válido. Dado que el evento `Paint` se producirá de inmediato, en cuanto la ventana se haga visible, esto podría causar un fallo del programa. Por ello comenzamos comprobando si `Imagen` está o no vacía. Si tiene una imagen, usamos el citado método para dibujarla en la ventana, concretamente una versión de `DrawImage()` que toma como segundo parámetro un punto a partir del cual dibujar. Ese punto lo obtenemos de la propiedad `AutoScrollPosition` del formulario, que es actualizado a medida que el usuario usa las barras de desplazamiento.

```
Private Sub Form1_Paint(ByVal sender As Object, _
ByVal e As System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint

  If Not (Imagen Is Nothing) Then

    e.Graphics.DrawImage(Imagen, Me.AutoScrollPosition)
  End If
End Sub
```

A pesar de la existencia de las barras de desplazamiento, tras recuperar una imagen es posible que el usuario de este programa modifique las dimensiones de la ventana, reduciéndolas o ampliándolas, para ver mejor la imagen. En ese momento se producirá un evento `Resize`, en cuyo método asociado nos limitaremos a llamar al método `Invalidate()` asegurando así que el contenido mostrado en cada momento en la ventana es el correcto.

Nota

También podemos activar el estilo `ResizeRedraw` del formulario, utilizando el método `SetStyle()`, si queremos que la invalidación se produzca automáticamente cuando se altere el tamaño de la ventana.

En este momento el programa ya sería capaz de recuperar una imagen cualquiera y mostrarla. Además, vamos a añadir en el método asociado al evento `KeyPress` el código necesario para rotar la imagen y añadir un título. El primer proceso se asociará a la pulsación de la letra `R` y el segundo a la letra `T`:

```
Private Sub Form1_KeyPress(ByVal sender As Object, _
ByVal e As System.Windows.Forms.KeyPressEventArgs) _
Handles MyBase.KeyPress

  If Char.ToUpper(e.KeyChar) = "R" Then

    Imagen.RotateFlip(RotateFlipType.Rotate90FlipNone)
    Invalidate()
  End If

  If Char.ToUpper(e.KeyChar) = "T" Then
    Dim Grafico As Graphics
    Dim TipoLetra As Font

    Grafico = Graphics.FromImage(Imagen)

    TipoLetra = New Font("Courier New", 24, _
        FontStyle.Bold Or FontStyle.Italic)

    Grafico.DrawString("GDI+", TipoLetra, Brushes.Blue, 10, 10)
    Invalidate()
  End If
End Sub
```

Con este código ya tenemos completo nuestro programa de visualización de imágenes, no tiene más que ejecutarlo y recuperar cualquier imagen para comprobar su funcionamiento. En la figura 13.14 puede verlo trabajando sobre una imagen del Tech•Ed 2001 que tuvo lugar en Barcelona.

Figura 13.14. El programa mostrando una imagen tras añadir unos títulos

Nota

En el ámbito `System.Drawing.Imaging` encontrará, aparte de la clase `Image` y la clase `Bitmap`, la clase `Metafile`. Con ella puede usar archivos que, en lugar de almacenar información sobre los puntos del gráfico o imagen, contienen los comandos, coordenadas y parámetros para generar la imagen.

Caminos y regiones

Aparte de múltiples métodos para el dibujo de entidades que podríamos denominar *sencillas*, como líneas, rectángulos, arcos y polígonos regulares e irregulares, en GDI+ podemos conseguir elementos más complejos utilizando caminos. Éstos pueden estar formados por líneas, rectas y curvas, o bien elementos obtenidos a partir de los métodos antes mencionados.

Los caminos, una vez creados, pueden dibujarse, obteniéndose tan sólo el contorno, y también rellenarse. Para ellos se utilizan los métodos `DrawPath()` y `FillPath()`, respectivamente, así como objetos que ya conocemos: pinceles y brochas. Al igual que muchos otros elementos, los caminos pueden ser transformados de diversas formas: rotados, escalados, trasladados, etc.

En cuanto a las regiones, también pueden estar formadas por entidades básicas o, incluso, por caminos, si bien su finalidad, respecto a los caminos, es muy distinta. No sirven para ser dibujadas exclusivamente, sino que se utilizan para limitar el área de dibujo, detectar la existencia de un punto en una determinada área o establecer la apariencia de una ventana no rectangular.

Las regiones pueden combinarse entre sí efectuando distintas operaciones, como la unión y la intersección, que dan una nueva región como resultado.

Creación de un camino

Un camino puede estar formado por una o más figuras, cada una de las cuales, a su vez, se forma con entidades simples: líneas, arcos, rectángulos, etc. Los métodos `StartFigure()` y `CloseFigure()` marcan el inicio y fin, respectivamente, de cada figura. Entre las llamadas a estos métodos se incluirán otras para formar la figura.

Las entidades básicas se añaden como parte del camino mediante los métodos `AddLine()`, `AddArc()`, `AddRectangle()` y similares. Son equivalentes a los métodos `DrawLine()`, `DrawArc()` y `DrawRectangle()` de la clase `Graphics`, con la diferencia de que esas entidades no se dibujan sino que se añaden como parte de una figura. Una alternativa son los métodos `AddCurve()`, `AddClosedCurve()` o `AddBeziers()`. Todos ellos toman como parámetro un arreglo de estructuras `Point` indicando los puntos por los que debe pasar la curva.

Finalizada la composición del camino, no tenemos más que usar los citados métodos `DrawPath()` y `FillPath()` para dibujar su contorno o rellenar su interior sobre una superficie de dibujo. El código siguiente, por ejemplo, añadido al evento `Paint` de un formulario genera el resultado que puede verse en la figura 13.15. La primera figura se compone de un rectángulo y una elipse, mientras que la segunda es una curva abierta.

```
Dim Camino As GraphicsPath = New GraphicsPath()

Dim Puntos() As Point = { _
      New Point(50, 150), _
      New Point(75, 200), _
      New Point(120, 140)}

With Camino
  .StartFigure()

  .AddRectangle(New Rectangle(10, 10, 100, 100))

  .AddArc(50, 75, 100, 50, 0, 359)
```

```
  .CloseFigure()
  .StartFigure()

  .AddCurve(Puntos)
End With

e.Graphics.FillPath(Brushes.Aqua, Camino)

e.Graphics.DrawPath(Pens.Blue, Camino)
```

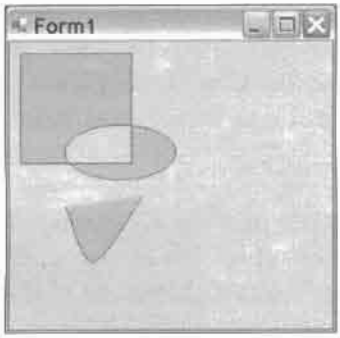


Figura 13.15. Aspecto del camino con sus dos figuras

Transformaciones

Los caminos, al igual que otros elementos gráficos, pueden sufrir diversas transformaciones que alteran su posición en el espacio, orientación y dimensiones. Estas transformaciones se efectúan mediante sumas y multiplicaciones de matrices, un tema relativamente complejo del cual, por suerte, no tenemos que preocuparnos ya que la clase `Matrix` se encarga de ello.

Una vez hemos creado un objeto de la clase `Matrix` básicamente podemos usar tres métodos distintos: `Scale()`, `Translate()` y `Rotate()`. Con el primero se alteran las proporciones originales del objeto que va a transformarse, con el segundo se cambia su posición en la superficie de dibujo y con el tercero, como puede suponer, se efectúa una rotación.

Los parámetros son fáciles de comprender en todos los casos: factores de escalado en el eje X e Y, desplazamiento en dichos ejes respecto a la posición original y grados que quiere rotarse.

Tomando como base el código del ejemplo anterior, podríamos añadir las sentencias siguientes al final para dibujar el mismo camino pero con un 75% de su tamaño original, algo más a la derecha y abajo y rotado 45°. El resultado sería el de la figura 13.16.

```
Dim Matriz As Matrix = New Matrix()

Matriz.Translate(200, 50)
Matriz.Scale(0.75, 0.75)
```

```
Matriz.Rotate(45)

Camino.Transform(Matriz)

e.Graphics.DrawPath(Pens.Red, Camino)
```

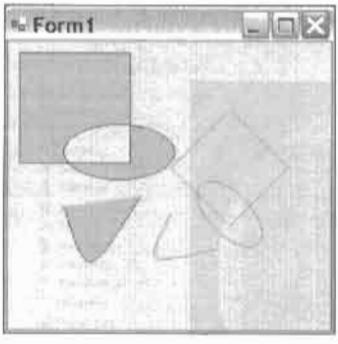


Figura 13.16. La ventana con el camino original y el transformado

Creación de una región

Las regiones son objetos de la clase `Region` y pueden ser creados a partir de rectángulos, objetos `GraphicsPath` y otras regiones. Una vez creada, a la región pueden añadirse otras mediante el método `Union()`. También puede crearse la intersección con otras regiones.

A diferencia de los caminos, las regiones no se dibujan en una superficie. Son objetos que mantienen un área, simple o compleja según los casos, que puede ser utilizada, por ejemplo, para limitar la zona en que puede dibujarse, comprobar si un punto existe en esa área o alterar la apariencia rectangular por defecto de las ventanas. Este último caso es, sin duda, el más llamativo.

Podemos ver un ejemplo, muy sencillo, del uso de regiones añadiendo el código siguiente en el método asociado al evento `Load` del formulario, consiguiendo una ventana circular con una pequeña área rectangular en la parte superior izquierda. Observe cómo se establece la apariencia del formulario mediante la propiedad `Region`.

```
Dim Rectangulo As Rectangle = New Rectangle( _
          0, 0, Me.Size.Width, Me.Size.Height)

Dim Camino As GraphicsPath = New GraphicsPath()

Camino.AddArc(Rectangulo, 0, 359)

Dim Area As Region = New Region(Camino)
```

```
Rectangulo.Width = 100
Rectangulo.Height = 40

Area.Union(Rectangulo)

Me.Region = Area
```

El efecto, obviamente, puede ser mucho más impactante utilizando una región con forma más elaborada, por ejemplo a partir de una imagen.

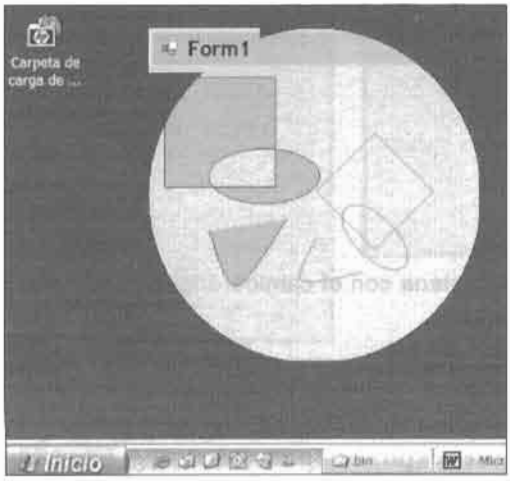


Figura 13.17. Aspecto de la ventana circular sobre el escritorio de Windows XP

El *componente* `PictureBox`

En todos los ejemplos de este capítulo hemos usado la superficie del formulario como lienzo para nuestros dibujos, utilizando siempre el evento `Paint` para mantener adecuadamente la composición. También hemos visto que podemos usar un objeto `Bitmap` con el objetivo de dibujar en memoria, sin necesidad de hacer visible la composición. Existen otras alternativas y una de ellas es el uso del control `PictureBox`.

Este componente, que podemos insertar visualmente en un formulario como cualquier otro control, dispone de una propiedad, llamada `Image`, a la que podemos asignar cualquier imagen, tanto en ejecución como durante la fase de diseño. Mediante la propiedad `SizeMode` podemos ajustar la imagen a las dimensiones del control o viceversa, según interese, mientras que con `BorderStyle` estableceremos el borde que deseamos aparezca alrededor. En la figura 13.18 puede ver cómo se ha recuperado una imagen en fase de diseño, imagen que aparece en el interior del `PictureBox`.

Si nos interesa dibujar en la superficie del `PictureBox` tenemos que usar el método `CreateGraphics()`. Este devuelve un objeto de la clase `Graphics`,

la misma que hemos utilizado repetidamente en los ejemplos previos. La única diferencia es que ese objeto ahora representa una superficie que está limitada a un control, no la superficie completa del formulario.

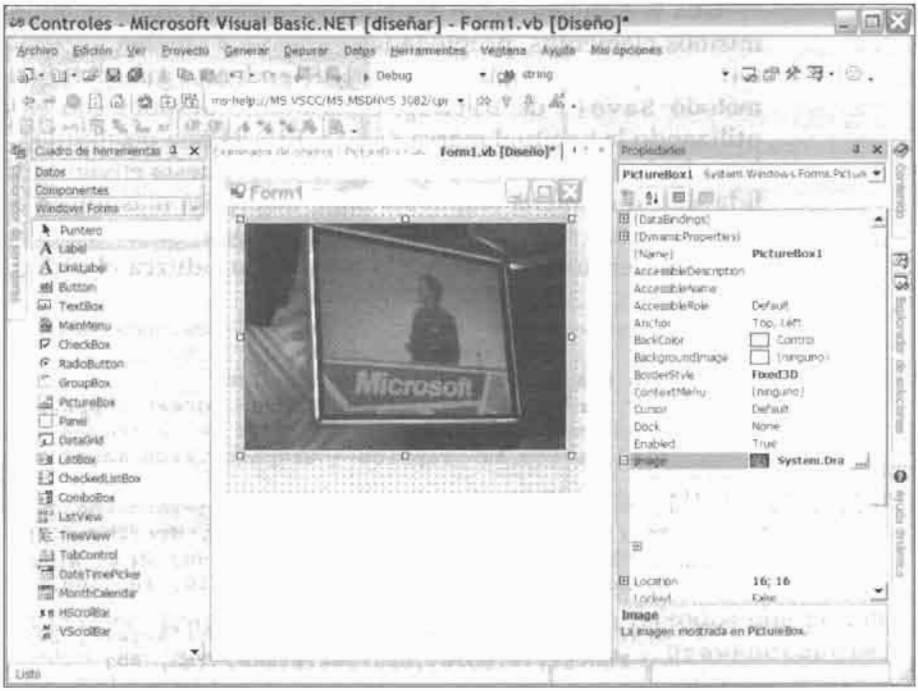


Figura 13.18. Un control `PictureBox` con una imagen en su interior

Gráficos en ASP.NET

Generalmente estamos acostumbrados a que las interfaces nativas, las basadas en formularios Windows, sean más ricas, visualmente hablando, que las interfaces basadas en la Web. En la práctica, sin embargo, la mayoría de los elementos que utilizamos en las primeras pueden también aplicarse a las segundas, si bien con una metodología diferente.

Suponga que está preparando una aplicación web y desea incluir imágenes en algunas páginas, o que necesita preparar un gráfico en respuesta a unos datos introducidos por el cliente de la aplicación. Son casos en los que GDI+ puede utilizarse conjuntamente con ASP.NET para alcanzar el objetivo deseado. Los pasos básicos son los siguientes:

- Preparación del gráfico en memoria, utilizando para ello un objeto de la clase `Bitmap`.
- Usando el método `FromImage()` de la clase `Graphics` podremos dibujar sobre ese objeto sin necesidad de que esté visible.

- Guardar el gráfico en un flujo en memoria, a fin de insertarlo en la respuesta al cliente, o bien escribirlo directamente en dicha respuesta usando la propiedad `OutputStream` de la clase `HttpResponse`.

Una vez que hayamos obtenido el objeto `Graphics` podemos usar todos los mismos elementos, propiedades y métodos que hemos conocido a lo largo de todo el capítulo. El gráfico resultante podemos guardarlo en un archivo, con el método `Save()` de `Bitmap`, enlazándolo después con el documento HTML utilizando la habitual marca `<img>`. Ésta es una posibilidad, la otra consiste en introducir directamente el gráfico en la respuesta enviada al cliente, opción válida sólo si en dicha respuesta no se incorpora más información.

Para ver un sencillo ejemplo inicie un nuevo proyecto de aplicación ASP.NET y, tras hacer doble clic sobre la página, introduzca el código siguiente:

```
' Creamos una imagen con unas dimensiones de 320 200
' y una profundidad de color de 32 bits
Dim Imagen As New Bitmap(320, 200, _
     Drawing.Imaging.PixelFormat.Format32bppRgb)
' obtenemos un Graphics a partir de la imagen
Dim Grafico As Graphics = Graphics.FromImage(Imagen)

' En este caso, pintamos toda la superficie con azul
Grafico.FillRectangle(Brushes.Blue, Grafico.ClipBounds)
   ' un elipse en la superficie
Grafico.FillEllipse(Brushes.Aqua, 10, 10, 300, 180)
  ' un texto
Grafico.DrawString("GDI+", New Font("Arial", 20, _
   FontStyle.Bold),Brushes.Black, 140, 80)

' enviamos la respuesta al cliente
Imagen.Save(Response.OutputStream, ImageFormat.Jpeg)
```

Observe cómo al crear el objeto `Bitmap` indicamos las dimensiones del gráfico y la profundidad de color. Para incluirlo en la respuesta enviada al cliente facilitamos al método `Save()` de `Bitmap` dos parámetros: la propiedad `OutputStream`, un objeto `Stream` que representa el flujo de datos enviado como respuesta, y una constante indicando el formato en que se codificará la imagen. El resultado es el que puede verse en la figura 13.19: un gráfico como respuesta a una solicitud. En la práctica el gráfico podría ser la representación de unos datos numéricos, una fotografía o cualquier otro mapa de bits.

Puntos clave

- Los servicios de GDI+ están repartidos por varios ámbitos, todos ellos parten de la raíz `System.Drawing`.
- La superficie de dibujo, indistintamente de que el dispositivo final sea una ventana, un mapa de bits en memoria o una impresora, está representada por la clase `Graphics`.

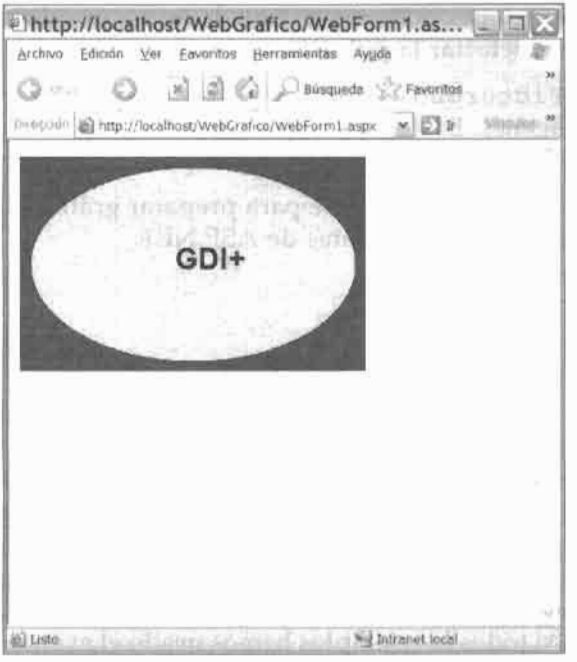


Figura 13.19. Un gráfico GDI+ en una página ASP.NET

- En la clase `Graphics` encontramos multitud de métodos que permiten dibujar todo tipo de entidades gráficas: `DrawLine()`, `DrawRectangle()`, `FillEllipse()`, etc.
- La gestión del color queda en manos de la estructura `Color` y sus propiedades y métodos. Existen decenas de colores predefinidos en forma de constantes.
- El lápiz o pincel con que se dibuja el contorno de las entidades gráficas está representado por la clase `Pen`. Por medio de sus propiedades podemos establecer el color, tipo de trazo, grosor, etc.
- Para los rellenos de superficies se utiliza una brocha cuyos atributos se definen mediante un objeto de tipo `Brush`. Podemos usar brochas sólidas, con tramas, degradados de color e incluso imágenes.
- Las entidades se sitúan en el plano usando un sistema de coordenadas lógico en el que podemos alterar el origen, la unidad de medida y el escalado. Dicho sistema es traducido al sistema final del dispositivo por GDI+.
- Mediante las clases derivadas de `Image` se facilita la recuperación, manipulación en memoria y almacenamiento de imágenes, como pueden ser los mapas de bits y los meta-archivos.
- Para la composición de entidades complejas podemos usar los caminos, objetos que se componen de entidades más simples y que posteriormente pueden ser transformados antes de dibujarse en el `Graphics`.

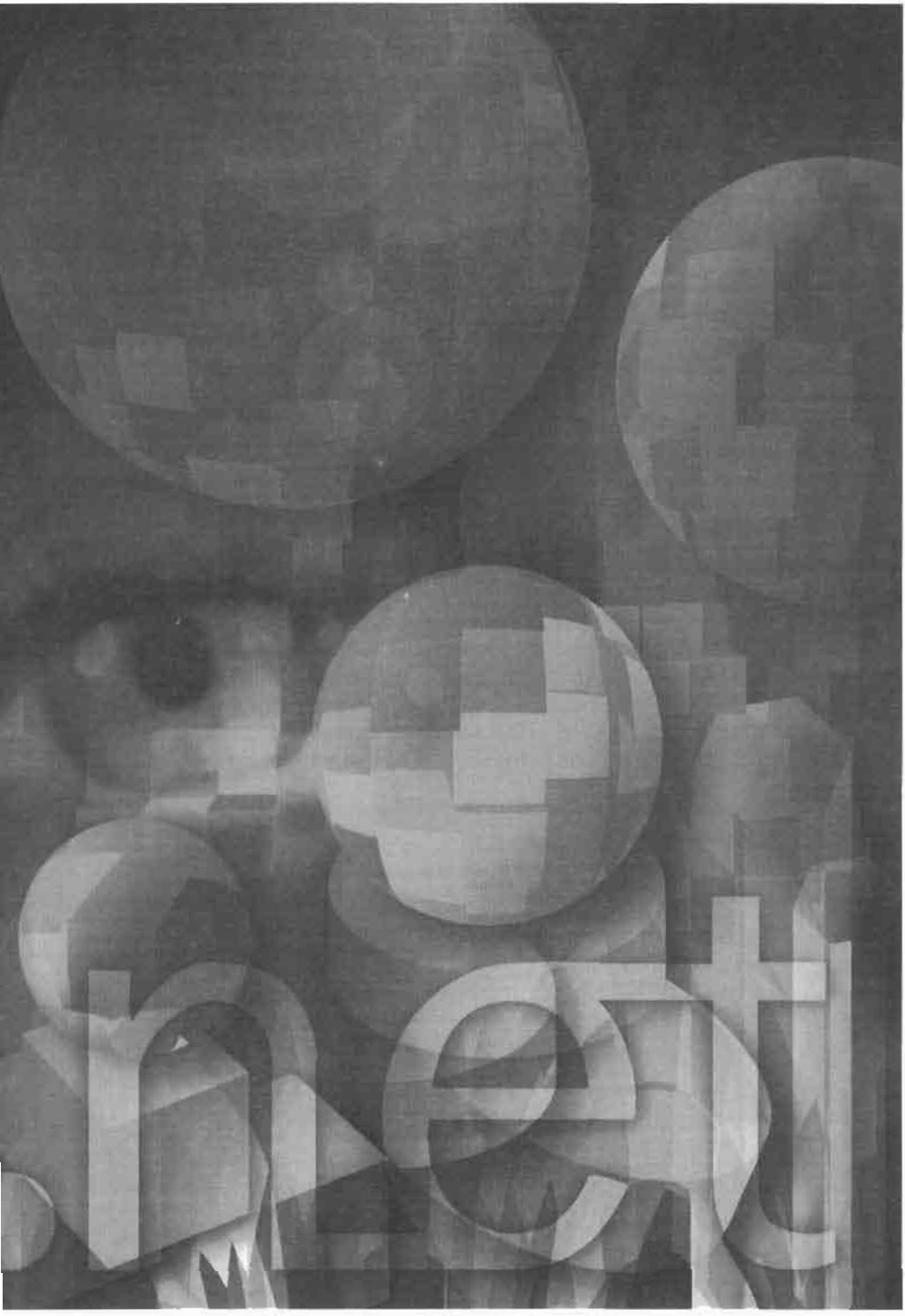
- Utilizando regiones podemos alterar la forma por defecto de una ventana, así como detectar la pulsación sobre áreas de un gráfico.
- El control `PictureBox` es un componente especializado en la visualización de imágenes. Basta una asignación a su propiedad `Image` para mostrar una imagen cualquiera, sin ningún paso adicional.
- GDI+ también puede utilizarse para preparar gráficos en memoria con el objetivo de enviarlos a clientes de ASP.NET.

Resumen

Este capítulo ha introducido sólo algunas de las posibilidades gráficas de GDI+, un modelo mucho más simple y flexible de trabajo con gráficos que el clásico GDI o las sentencias originales que Visual Basic heredó de las versiones ancestrales de BASIC. Utilizando los lápices, brochas y los métodos de la clase `Graphics` ya tenemos más posibilidades que en versiones previas de Visual Basic. A esto se añaden las capacidades para la manipulación de imágenes, los caminos y regiones o las diversas transformaciones.

Aunque en casi todos los ejemplos hemos usado el evento `Paint` para dibujar sobre la superficie del formulario, lo cierto es que podemos dibujar directamente sobre un `Bitmap`, lo hemos visto al final, y, por ejemplo, conseguir efectos de animación mediante la técnica conocida como *doble buffer*. También se pueden registrar las sentencias en un meta-archivo y, posteriormente, reproducirlo sobre la superficie de dibujo.

Por último, también forman parte de GDI+ los servicios relativos a impresión. De ellos exclusivamente nos ocuparemos en el próximo capítulo.



14 Impresión de datos

La mayoría de las aplicaciones que desarrollemos procesarán información que, de alguna forma, será almacenada persistentemente, por regla general en bases de datos.

Esa información, no obstante, una vez procesada tendrá unos destinatarios que no deben, necesariamente, usar la misma aplicación. En estos casos surge la necesidad de publicar la información ya elaborada, usando para ello recursos como la impresión en papel o la publicación digital.

En Visual Basic .NET podemos usar una serie de componentes que facilitan la salida de información por la impresora, así como la visualización previa de esa impresión. Será el tema central de este capítulo.

El destino para esa información, en realidad, no tiene necesariamente que ser una impresora física, ya que en Windows podemos configurar como dispositivo de salida un fax, un dispositivo virtual para generar documentos PDF o un archivo.

Una alternativa al control directo de la impresión de los datos, aplicable si contamos con una de las ediciones superiores de Visual Studio .NET o bien hemos adquirido por separado Crystal Reports.NET, consiste en usar esta conocida herramienta para diseñar los informes, imprimirlos y exportarlos en diversos formatos.

Crystal Reports.NET se integra en el entorno de Visual Studio.NET aportando su propio diseñador y componentes que simplifican su uso. Trataremos el uso de esta herramienta en un capítulo posterior.

Fases del proceso de impresión

Comencemos analizando los pasos más básicos de que consta el proceso de impresión en una aplicación Windows escrita con Visual Basic .NET. Este proceso es siempre básicamente el mismo, si bien después podemos añadir otras posibilidades como la configuración de página por parte del usuario o el ofrecimiento de una visualización previa.

En el sistema siempre existe una impresora seleccionada como dispositivo de salida por defecto. En principio la usaremos como destino de nuestra información, aunque después veremos cómo facilitar la selección de una impresora y su configuración. Tendrá, por tanto, que conectar esa impresora por defecto para ver el resultado generado por los distintos ejemplos.

Los datos a imprimir pueden, virtualmente, proceder de cualquier fuente: una base de datos, un archivo de texto, los elementos de un arreglo o, simplemente, aparecer como constantes en el interior del programa. Por simplicidad, para centrarnos principalmente en las tareas de impresión, usaremos datos constantes. Éstos podrán sustituirse, por ejemplo cuando aprenda a usar bases de datos, por información procedente de otras fuentes.

Documento a imprimir

Los elementos a los que se da salida por un dispositivo de impresión se denominan documentos, estando representados por un componente `PrintDocument`. Encontrará este componente en el Cuadro de herramientas, en concreto al final de la página Windows Forms. Se trata de un componente no visual que al ser insertado en un formulario aparece en la parte inferior del diseñador. La única propiedad interesante que encontrará en la ventana Propiedades es `DocumentName`. Con ella se establece el nombre del documento, el nombre de cara al usuario que aparecerá, por ejemplo, en la ventana de la impresora.

Para poner en marcha el proceso de impresión tan sólo es imprescindible un paso: la llamada al método `Print()` del componente `PrintDocument`. A partir de ese momento se desencadenará una secuencia de eventos que nos permitirán controlar el proceso y facilitar la información que debe imprimirse. Estos eventos son:

- `BeginPrint` – Se produce una sola vez, inmediatamente después de llamar a `Print()`. Es el lugar adecuado para preparar los datos que van a imprimirse a continuación, por ejemplo conectando con la base de datos, abriendo el archivo del que van a obtenerse, etc.
- `QueryPageSettings` – Este evento se genera una vez por cada página que va a imprimirse, concretamente justo antes de que se imprima dicha página. El método que lo reciba contará con un parámetro, un objeto de la clase `PageSettings`, que puede usar para configurar la página. Volveremos sobre ellos posteriormente.

- `PrintPage` – Al igual que el anterior, se genera una vez por cada página a imprimir. Es el único evento que debe controlarse obligatoriamente, ya que es el encargado de facilitar la información a imprimir e informar si hay más páginas pendientes.
- `EndPrint` – Complementario a `BeginPrint`, se produce una vez ha finalizado la impresión de la última página. Es, por tanto, el punto adecuado para liberar recursos, cerrar conexiones abiertas, etc.

En la práctica, asumiendo el caso más simple posible, incluiríamos un componente `PrintDocument` en el formulario, llamaríamos a su método `Print()` desde algún punto del programa y nos serviríamos del evento `PrintPage` para preparar cada página.

Superficie de impresión

Cada vez que se produce el evento `PrintPage`, el componente `PrintDocument` facilita un parámetro de tipo `PrintPageEventArgs`. En él encontramos, por una parte, las dimensiones de la página sobre la que va a imprimirse, totales y aplicando los márgenes preestablecidos, así como un objeto `Graphics` sobre el que dibujaremos el contenido de la página.

La impresión, por tanto, se hace sobre una superficie representada por un objeto `Graphics` que, básicamente, ya conocemos por haberlo usado en el capítulo previo. Podemos usar el método `DrawString()` para imprimir el texto, controlando su apariencia mediante la clase `Font`. Igualmente, podemos imprimir gráficos utilizando los métodos de la clase `Graphics`.

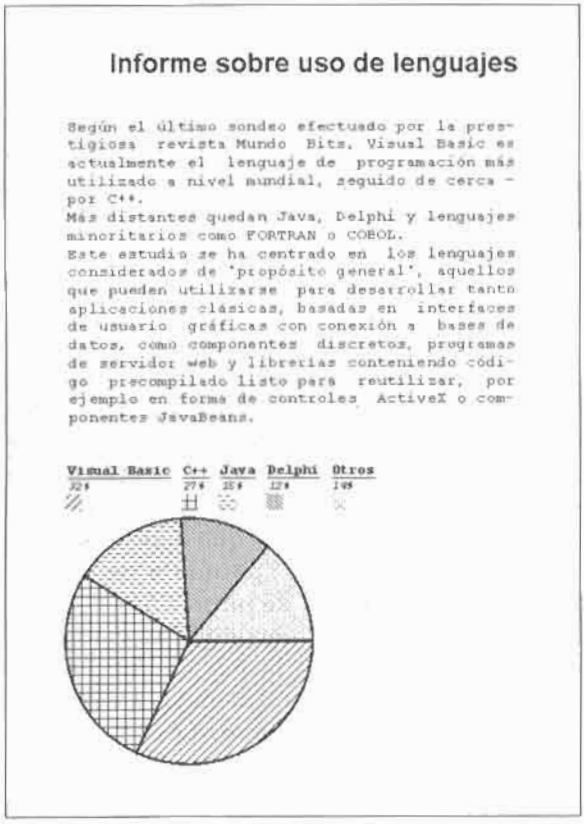
Debemos tener en cuenta el contenido de las propiedades `PageBounds` y `MarginBounds`. Ambas son estructuras `Rectangle`, la primera conteniendo las dimensiones de la página y la segunda con las coordenadas que quedan dentro de los márgenes, donde se supone que debemos imprimir. También podemos usar la propiedad `VisibleClipBounds` de la clase `Graphics` para obtener el rectángulo visible del área de impresión.

Antes de devolver el control, desde el método asociado al evento `PrintPage`, nuestro programa habrá completado la página con los datos a imprimir. Si queda más información para imprimir daremos el valor `True` a la propiedad `HasMorePages` del objeto `PrintPageEventArgs` que se recibe como parámetro. Sólo con esto, ya podemos componer un documento con las páginas que sean necesarias. Normalmente necesitaremos controlar la posición vertical en la que imprimiremos, incrementándola según el texto o gráficos que vayan introduciéndose. Cuando esa posición esté cerca del final devolvemos el control.

Nota

Si va a imprimir sólo texto en la página, para calcular el número de líneas disponibles divida `MarginBounds.Height` entre la altura del texto, dato que puede obtener mediante el método `GetHeight()` de `Font`.

Tomando como base los elementos vertidos en los puntos previos, veamos cómo compondríamos para imprimir un documento de relativa complejidad. Éste estará formado por un título, un texto, una pequeña tabla de datos y un gráfico simple que los representa. Toda esta información, como ya se ha dicho, podría recuperarse de un archivo o una base de datos pero, en nuestro caso, la obtendremos a partir de diversos arreglos. Al completar la impresión, obtendremos en el papel algo parecido a lo que se muestra en la figura 14.1. Si hay más texto, el gráfico, o incluso parte del texto junto con la tabla de datos, pasarían a las páginas siguientes.



Informe sobre uso de lenguajes

Según el último sondeo efectuado por la prestigiosa revista Mundo Bits, Visual Basic es actualmente el lenguaje de programación más utilizado a nivel mundial, seguido de cerca por C++.
Más distantes quedan Java, Delphi y lenguajes minoritarios como FORTRAN o COBOL.
Este estudio se ha centrado en los lenguajes considerados de "propósito general", aquellos que pueden utilizarse para desarrollar tanto aplicaciones clásicas, basadas en interfaces de usuario gráficas con conexión a bases de datos, como componentes discretos, programas de servidor web y librerías conteniendo código precompilado listo para reutilizar, por ejemplo en forma de controles ActiveX o componentes JavaBeans.

Visual Basic	C++	Java	Delphi	Otros
32%	27%	25%	12%	1%

Figura 14.1. Aspecto de la página una vez impresa

Comenzaremos añadiendo al inicio de la definición de la clase, correspondiente al formulario, una serie de arreglos con los datos a imprimir. Concretamente tendremos los siguientes miembros:

```
' Título del informe
Private Titulo As String = _
        "Informe sobre uso de lenguajes"
```

```
Private Titulos() As String = { _
   "Visual Basic", "C++", "Java", _
   "Delphi", "Otros"}

Private Datos() As String = { _
   "32%", "27%", "15%", "12%", "14%"}

Private Puntos(,) As Single = { _
  {0, 32 * 3.6}, {32 * 3.6, 27 * 3.6}, _
  {(32 + 27) * 3.6, 15 * 3.6}, _
  {(32 + 27 + 15) * 3.6, 12 * 3.6}, _
  {(32 + 27 + 15 + 12) * 3.6, 14 * 3.6}}

Private Relleno() As HatchStyle = { _
  HatchStyle.BackwardDiagonal, _
  HatchStyle.Cross, HatchStyle.DashedHorizontal, _
  HatchStyle.LightDownwardDiagonal, _
  HatchStyle.DottedDiamond}

Private LineaTexto As Byte

Private Texto() As String = { _
 "Según el último sondeo efectuado por la pres-", _
 "tigiosa  revista Mundo  Bits, Visual Basic es", _
 "actualmente el  lenguaje de  programación más", _
 "utilizado a nivel mundial, seguido de cerca -", _
 "por C++.", _
 "Mas distantes quedan Java, Delphi y lenguajes", _
 "minoritarios como FORTRAN o COBOL.", _
 "Este estudio se ha centrado en  los lenguajes", _
 "considerados de 'propósito general', aquellos", _
 "que pueden utilizarse  para desarrollar tanto", _
 "aplicaciones clásicas, basadas en  interfaces", _
 "de usuario  gráficas con conexión a  bases de", _
 "datos, como componentes  discretos, programas", _
 "de servidor web y librerías conteniendo códi-", _
 "go  precompilado listo para  reutilizar,  por", _
 "ejemplo en forma de controles  ActiveX o com-", _
 "ponentes JavaBeans."}
```

Copiando y pegando líneas podemos aumentar la cantidad de texto del informe, o bien reducirlo, con el fin de comprobar cómo se producen los saltos de una página a otra gracias a la lógica que vamos a analizar de inmediato.

La variable `LineaTexto` es fundamental, ya que mediante ella sabremos si estamos iniciando el informe, qué líneas se han impreso o si se ha finalizado con el texto. El valor inicial de esta variable será cero. Asumiendo que ha iniciado una nueva aplicación Windows, inserte en el formulario un componente `PrintDocument` y un botón. Haga luego doble clic sobre éste y llame al método `Print()` del primero, de tal manera que la impresión se inicie a demanda,

cuando pulsemos ese botón. El resto del código estará en el método `PrintPage` del componente `PrintDocument`, código que vamos a estudiar ahora paso a paso.

Lo primero que tenemos en ese método son los tipos de letra, brocha y lápiz que vamos a utilizar para imprimir las distintas partes del texto y el gráfico. La definición de estos elementos es la mostrada a continuación:

```
Dim TipoTexto As Font = New Font("Courier New", 14, _
      FontStyle.Regular)
Dim TipoTitulo As Font = New Font("Arial", 24, FontStyle.Bold)
Dim TipoTitulosDatos As Font = New Font("Courier New", 12, _
      FontStyle.Underline Or FontStyle.Bold)
Dim TipoDatos As Font = New Font("Courier New", 10, _
      FontStyle.Italic)


Dim Brocha As Brush
Dim Lapiz As Pen = New Pen(Color.Black, 3)
```

Como puede ver, hay un tipo de letra para el título del informe, otro para el texto general, un tercero para los títulos de los datos y un cuarto para los datos propiamente dichos.

El evento `PrintPage` se generará una primera vez, al llamarse al método `Print()`, cuando la variable `LineaTexto` tiene el valor 0. Comprobamos este hecho, que nos indica que el informe está comenzando a imprimirse, y lo aprovechamos para escribir el título centrado en la parte superior de la página. También damos un valor inicial a la variable `PosY`, que iremos actualizando para saber en qué posición de la página, verticalmente hablando, debemos comenzar a imprimir cada línea de texto.

```
If LineaTexto = 0 Then

  PosY = e.MarginBounds.Y

  Dim Tamano As SizeF = _
    e.Graphics.MeasureString(Titulo, TipoTitulo)


  e.Graphics.DrawString(Titulo, TipoTitulo, _
    Brushes.Black, (e.PageBounds.Width - _
    Tamano.Width) / 2, PosY)


  PosY += Tamano.Height * 2
End If
```

Para obtener el tamaño del recuadro necesario para imprimir el título usamos el método `MeasureString()` de la clase `Graphics`. Restando del ancho total de la página la anchura que ocupará el texto, y dividiendo el resultado entre dos, obtenemos el centro horizontal, punto en el que colocamos el título. Una

vez que `LineaTexto` no tenga el valor 0 el título no volverá a imprimirse, es decir, tan sólo aparecerá en la primera página.

Lo siguiente será comenzar a imprimir el texto del supuesto informe, texto que tenemos en un arreglo de cadenas. Para recorrerlo preparamos un índice y también calculamos el alto que tendrá cada línea de texto.

```
Dim Indice As Byte

Dim AltoLinea As Integer = TipoTexto.GetHeight(e.Graphics)
```

Con el fin de imprimir las líneas de texto que correspondan usamos un bucle. El índice de partida es el que indique `LineaTexto`, inicialmente cero, y el final el correspondiente a la última línea de texto del arreglo. Realmente el bucle, como puede verse en el código siguiente, puede terminar antes en caso de que se llegue al final de la página. En este caso conservamos el índice de la última línea en `LineaTexto`, asignamos el valor `True` a la propiedad `HasMorePages` y luego abandonamos el método. Cuando el evento `PrintPage` vuelva a generarse `LineaTexto` ya no será cero, de tal manera que no se imprimirá el título y se entrará en el bucle de impresión de líneas partiendo con el último índice que se usó al abandonarlo.

```
For Indice = LineaTexto To Texto.GetUpperBound(0)

  e.Graphics.DrawString(Texto(Indice), TipoTexto, _
        Brushes.Black, e.MarginBounds.Left, PosY)

  PosY += AltoLinea


  If PosY + AltoLinea >= e.MarginBounds.Bottom Then

    LineaTexto = Indice

    PosY = e.MarginBounds.Top

    e.HasMorePages = True
    Exit Sub
  End If
Next
```

El proceso de salida y entrada en el bucle puede repetirse múltiples veces, dependiendo de la cantidad de líneas de texto que haya en el arreglo, completando la impresión de múltiples páginas. Aunque es algo que en este caso no hemos hecho, podríamos usar un contador para saber en qué página estamos e imprimir ese dato como cabecera o pie.

Nos queda imprimir la tabla de datos y dibujar el correspondiente gráfico sectorial. Éste ocupará el espacio que quede libre entre dicha tabla y el final de la página, aunque manteniendo una proporción para aparecer siempre circular. Comenzaremos, por tanto, comprobando si hay espacio suficiente en la página, provocando un nuevo salto en caso contrario.

```
LineaTexto = Texto.GetUpperBound(0)+1

PosY += AltoLinea * 2

If PosY + TipoTitulosDatos.GetHeight(e.Graphics) _
      * 2 > e.MarginBounds.Bottom Then
  PosY = e.MarginBounds.Top
  e.HasMorePages = True
  Exit Sub
End If
```

Esta posible salida del método causará la generación de un nuevo evento `PrintPage`. En este caso ya no se imprimirá el título del informe y, gracias al valor que tiene `LineaTexto` se saltará sin ejecutar el bucle que imprime el texto. Se procederá a imprimir directamente la tabla de datos y el gráfico, un trabajo que hacemos con el código siguiente:

```
Dim PosX As Integer = e.MarginBounds.Left

AltoLinea = TipoTitulosDatos.GetHeight(e.Graphics)

For Indice = 0 To Titulos.GetUpperBound(0)
  e.Graphics.DrawString(Titulos(Indice), _
   TipoTitulosDatos, Brushes.Black, PosX, PosY)

  e.Graphics.DrawString(Datos(Indice), _
    TipoDatos, Brushes.Black, PosX, _
    PosY + AltoLinea)

  Brocha = New HatchBrush(Relleno(Indice), _
     Color.Black, Color.White)

  e.Graphics.FillRectangle(Brocha, PosX, _
     PosY + AltoLinea * 2, 20, 20)

  PosX += e.Graphics.MeasureString( _
    Titulos(Indice), TipoTitulosDatos).Width + 10
Next

PosY += AltoLinea * 2 + 30
PosX = e.MarginBounds.Left
```

```
Dim Alto As Single = e.MarginBounds.Height - PosY
Dim Ancho As Single = e.MarginBounds.Width

If Alto > Ancho Then Alto = Ancho
If Ancho > Alto Then Ancho = Alto

For Indice = 0 To Datos.GetUpperBound(0)

  Brocha = New HatchBrush(Relleno(Indice), _
    Color.Black, Color.White)

  e.Graphics.FillPie(Brocha, PosX, PosY, Ancho, Alto, _
    Puntos(Indice, 0), Puntos(Indice, 1))

  e.Graphics.DrawPie(Lapiz, PosX, PosY, Ancho, Alto, _
    Puntos(Indice, 0), Puntos(Indice, 1))
Next
```

Los porcentajes de cada lenguaje se colocan debajo de los títulos, formando así la pequeña tabla resumen del informe. Bajo cada dato se dibuja un rectángulo con el tipo de relleno que tendrá el correspondiente sector en el gráfico de tarta, facilitando así su identificación. Para dibujar los sectores utilizamos los métodos `DrawPie()` y `FillPie()` con los mismos parámetros de coordenadas. Éstas determinan el rectángulo, cuadrado en este caso, en que se incluirá el sector, mientras que los dos últimos parámetros determinan el punto de la circunferencia donde comenzará el arco y los grados de distancia hasta el punto final. Estos datos los tenemos precalculados en un arreglo que habíamos definido al principio. Dado que la suma de los porcentajes es 100 y la circunferencia 360 grados, es fácil hallar cada punto sumando los valores y multiplicándolos por 360/100, es decir, 3.6.

Para ejecutar el programa, y poder ver el resultado, deberá disponer de una impresora conectada al equipo. Pruebe a alterar la cantidad de texto del arreglo `Texto` para provocar más o menos saltos de página y ver el resultado.

Nota

Si dispone de una impresora en color, en lugar de usar brochas con diferentes patrones de relleno, como se hace en este ejemplo, puede utilizar un color distinto para cada sector.

Configuración de parámetros

En el ejemplo anterior se ha impreso un documento usando los parámetros de configuración predeterminados. Éstos fijan las dimensiones del papel, el

dispositivo de salida y algunos aspectos más, como el número de copias, el orden, etc. Visual Basic .NET cuenta con los cuadros de diálogo apropiados para permitir que el usuario establezca la mayoría de estos parámetros antes de realizar la impresión, ajustando ésta a sus preferencias.

Tomando como base el ejemplo anterior, cuyo formulario dispone de un botón que pone en marcha la impresión, vamos a ver cómo añadir algunas de estas posibilidades de configuración.

Selección del dispositivo de destino

No es extraño que en un mismo sistema haya instalados distintos sistemas de impresión: una impresora básica local, una impresora de red en color o de alto rendimiento, un dispositivo de fax, etc. Uno de esos dispositivos será el destino predeterminado para los trabajos de impresión en los que, de manera explícita, no se indique lo contrario.

El componente `PrintDocument` dispone de una propiedad, llamada `PrinterSettings`, que contiene todos los parámetros de configuración para impresión. Uno de los miembros de `PrinterSettings` contiene una colección con todos los dispositivos de salida existentes en el sistema. Esta propiedad es `InstalledPrinters`, mientras que `Printer` aloja el nombre de la impresora elegida como destino.

De manera análoga, utilizando las propiedades `PrinterResolutions`, `PaperSizes` y `PaperSources` de `PrinterSettings` podemos obtener una lista de todas las resoluciones que puede usar la impresora, los tamaños de papel contemplados y los alimentadores disponibles.

El número de copias que se quieren imprimir, el rango de páginas y su orden son factores controlables mediante las propiedades `Copies`, `FromPage`, `ToPage` y `Collate`, respectivamente.

Es posible limitar las páginas que el usuario puede elegir mediante las propiedades `MaximumPage` y `MinimumPage`, así como limitar el número de copias con `MaximumCopies`.

Aunque podríamos crear una interfaz de usuario para facilitar la configuración de todos estos parámetros, resulta mucho más cómodo utilizar un componente `PrintDialog`. Éste dispone también de la propiedad `PrinterSettings`, a la que podemos asignar los parámetros iniciales y de la cual recogeríamos los seleccionados por el usuario. El cuadro de diálogo se muestra con el habitual método `ShowDialog()`. Podemos comprobar el valor devuelto para saber si se ha aceptado la configuración.

Añada al formulario del ejemplo anterior los elementos que puede ver en la figura 14.2: dos controles `Label`, un `TextBox`, un `Button` y un `ListBox`. A continuación haga doble clic sobre el formulario, para abrir el método asociado al evento `Load`, e introduzca el código siguiente:

```
Dim Impresora As String
[illegible]
For Each Impresora In _
  PrintDocument1.PrinterSettings.InstalledPrinters
```

```
    ListBox1.Items.Add(Impresora)
Next

tbNombreImpresora.Text = PrintDocument1.PrinterSettings.PrinterName
```

Como puede ver, añadimos los nombres de todas las impresoras instaladas a la lista, mientras que en la caja de texto mostramos el nombre de la impresora seleccionada.

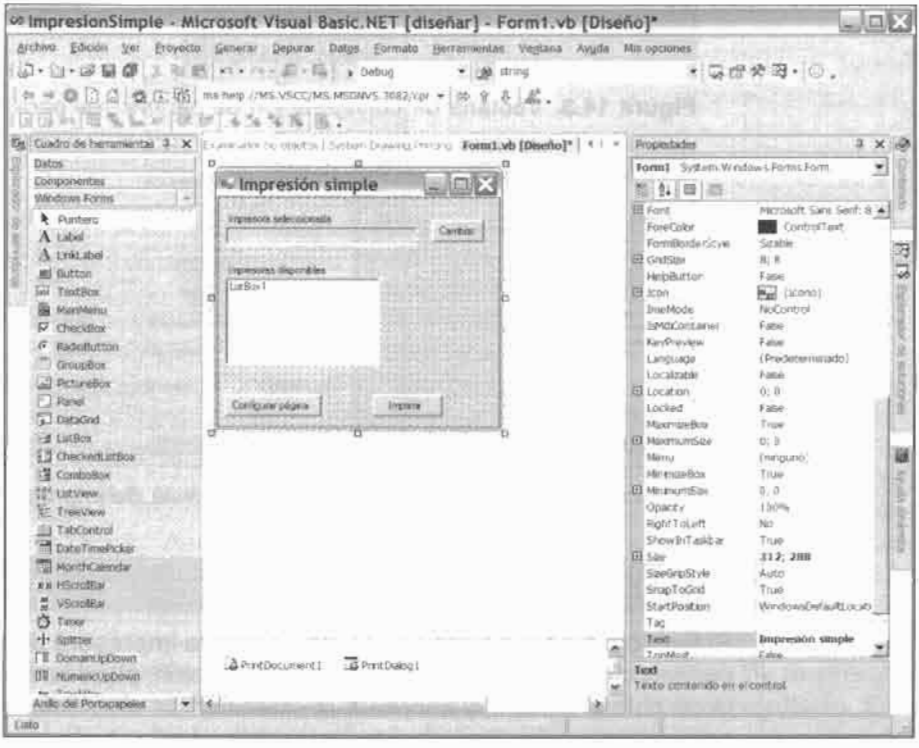


Figura 14.2. Aspecto del formulario tras insertar la lista y demás elementos

Inserte también un componente `PrintDialog`, que aparecerá en la parte inferior del diseñador, y use la ventana Propiedades para modificar su propiedad `Document`, relacionando este componente con el `PrintDocument` que ya tenemos. Esto facilita la sincronización automática entre la propiedad `PrinterSettings` del primero y el segundo, sin necesidad de ninguna intervención por nuestra parte.

Tan sólo nos queda escribir el código asociado al botón para cambiar de impresora, botón que abrirá el cuadro de diálogo que puede ver en la figura 14.3. Si cambia de impresora el nombre de ésta aparecerá en el formulario, como en la figura 14.4. Tenga en cuenta que los cambios que efectúe tendrán efecto sobre la impresión real, por lo que puede cambiar cualquier parámetro y a continuación pulsar el botón de impresión para ver el resultado.

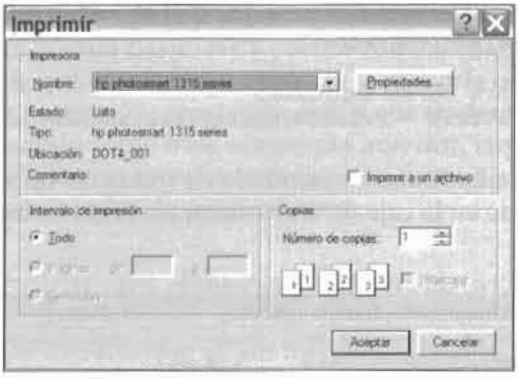


Figura 14.3. Ventana de configuración de parámetros para impresión

Figura 14.4. Aspecto del formulario con la lista de impresoras

Nota

Si tan sólo quiere permitir al usuario elegir la impresora de destino, sin acceder a los demás parámetros de configuración, puede usar una lista como la de este programa y controlar la selección de un elemento cuyo nombre se asignaría a la propiedad `PrinterSettings.PrinterName` del componente `PrintDocument`.

Configuración de página

Aparte del dispositivo de destino, el número de copias o el rango de páginas a imprimir, existen otros parámetros, como las dimensiones del papel o su orientación y márgenes, que también afectan a la impresión. Todos estos datos están almacenados en la propiedad `DefaultPageSettings`, cuyo tipo es `PageSettings`. Al igual que ocurre con `PrinterSettings`, podríamos recuperar y manipular de manera individual los datos de cada una de las propiedades de `PageSettings` pero, afortunadamente, ya existe un componente que ofrece un cuadro de diálogo para acceder a estos parámetros. Este componente es `PageSetupDialog`.

Después de insertar este componente en nuestro formulario, y asociarlo con el `PrintDocument` mediante la propiedad `Document`, no tenemos más que llamar al método `ShowDialog()` para mostrar el cuadro de diálogo de la figura 14.5. Automáticamente, la configuración que se efectúe será aplicada a la propiedad `DefaultPageSettings`.

Puede hacer una prueba insertando un componente `PageSetupDialog` en el formulario, añadiendo un nuevo botón y llamando al método `ShowDialog()` desde su evento `Click`.

Figura 14.5. Ventana para la configuración de página

Visualización preliminar

Muchas aplicaciones ofrecen, a la hora de realizar un trabajo de impresión, la posibilidad de ver previamente en pantalla cómo quedará el resultado. Esto facilita el ajuste de diversos parámetros, como la orientación o márgenes del papel, sin necesidad de imprimir físicamente en papel cada prueba que pueda hacerse. Cuando el aspecto es el deseado, generalmente desde la misma ventana de visualización preliminar existe algún botón u opción para enviar el documento a la impresora.

Ofrecer esa misma posibilidad en nuestros programas es realmente fácil, un trabajo que tan sólo requerirá insertar un componente, asignar valor a una propiedad y llamar a un método, así de simple. Puede comprobarlo insertando en el formulario un componente `PrintPreviewDialog`, asociándolo con el `PrintDocument` mediante la propiedad `Document` que hemos utilizado anteriormente en otros componentes, y sustituyendo después la llamada `PrintDocument1.Print()` que se producía al pulsar el botón de impresión por la llamada `PrintPreviewDialog1.ShowDialog()`.

Al pulsar dicho botón ahora no se imprimirá el documento, sino que aparecerá un cuadro de diálogo como el de la figura 14.6. Desde él puede ver sólo

una página o varias, ajustando su tamaño según le convenga. Si el resultado es satisfactorio pulsaría el botón que hay en la parte superior izquierda para imprimirlo, en caso contrario puede cerrar la ventana y modificar los parámetros de configuración de página o impresora, volviendo posteriormente a la visualización preliminar.

Si desea una previsualización más personalizada, en lugar del componente `PrintPreviewDialog` puede servirse del control `PrintPreviewControl`. Éste aparece como una página en la que se visualiza el documento a imprimir. No dispone de botones ni elementos adicionales, ya que su funcionamiento se controla mediante propiedades como `StartPage`, `Zoom`, `Columns` y `Rows`, estableciendo la primera página a mostrar, el nivel de magnificación, el número de páginas a mostrar horizontal y verticalmente. En la figura 14.7 puede ver un control de este tipo en el interior de un formulario. Lógicamente, tendríamos que añadir los elementos necesarios para cambiar la previsualización e imprimir el documento.

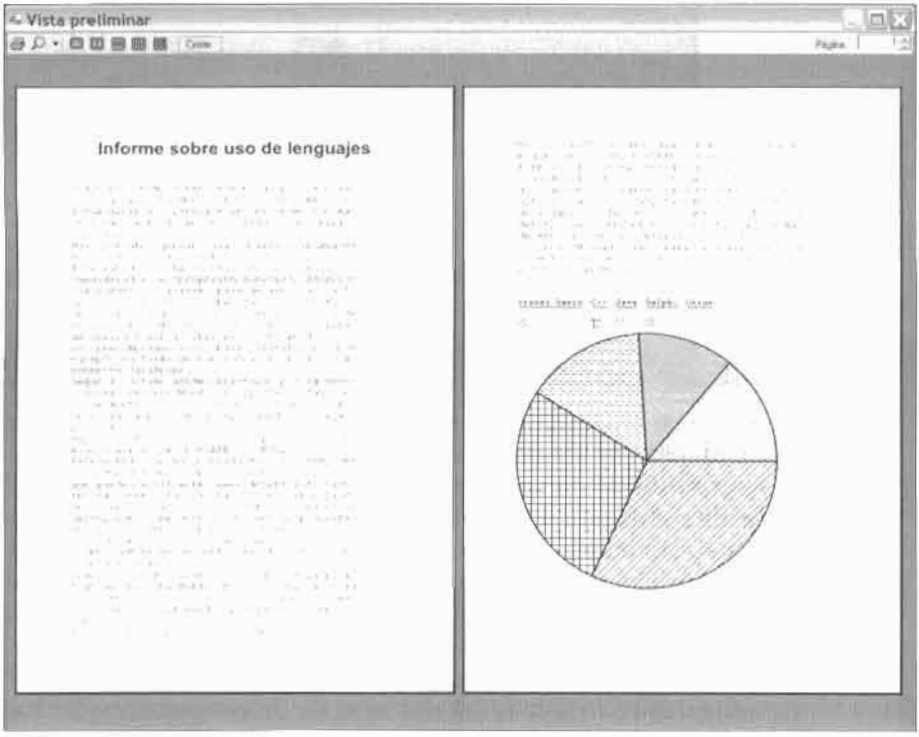


Figura 14.6. Visualización preliminar del documento a imprimir

Puntos clave

- La impresión de datos en Visual Basic .NET se gestiona usando un componente `PrintDocument`. Éste dispone de las propiedades, eventos y

métodos necesarios para configurar los parámetros de impresión y controlar el proceso.

- Para imprimir datos se utiliza la superficie de la página, representada por un objeto de la clase `Graphics`. Podemos usar, por tanto, todos los métodos que conocimos en el capítulo previo para elaborar gráficos.
- Existen cuadros de diálogo prediseñados, como `PrintDialog` y `PageSetupDialog`, que facilitan la configuración de los parámetros de impresión y formato de página.
- Mediante el cuadro de diálogo `PrintPreviewDialog` es posible facilitar una vista previa de la impresión. También puede utilizarse un control `PrintPreviewControl` para facilitar esa vista en una ventana propia.

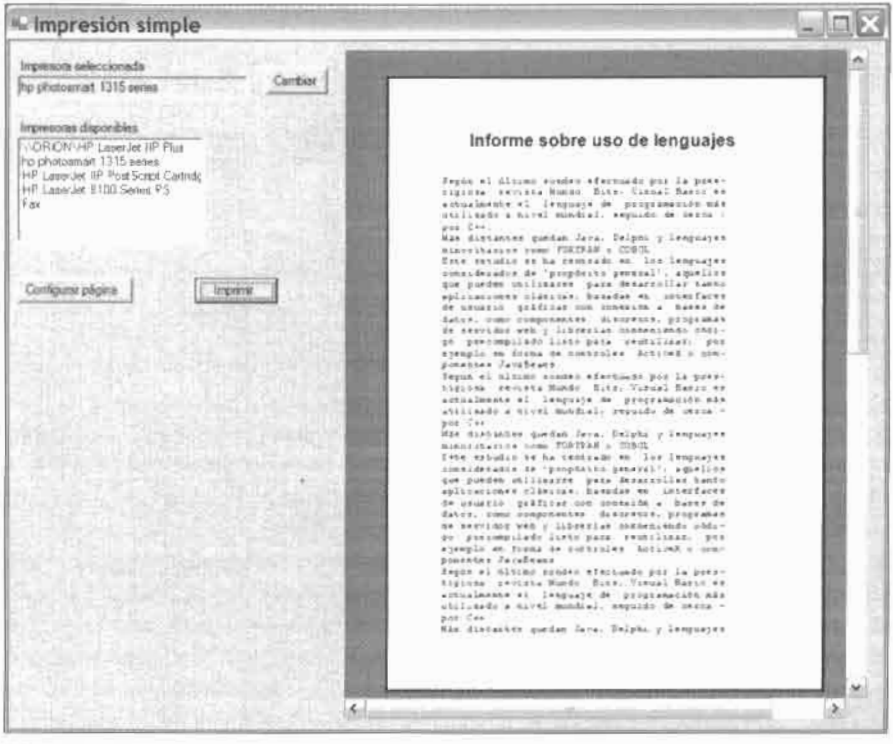


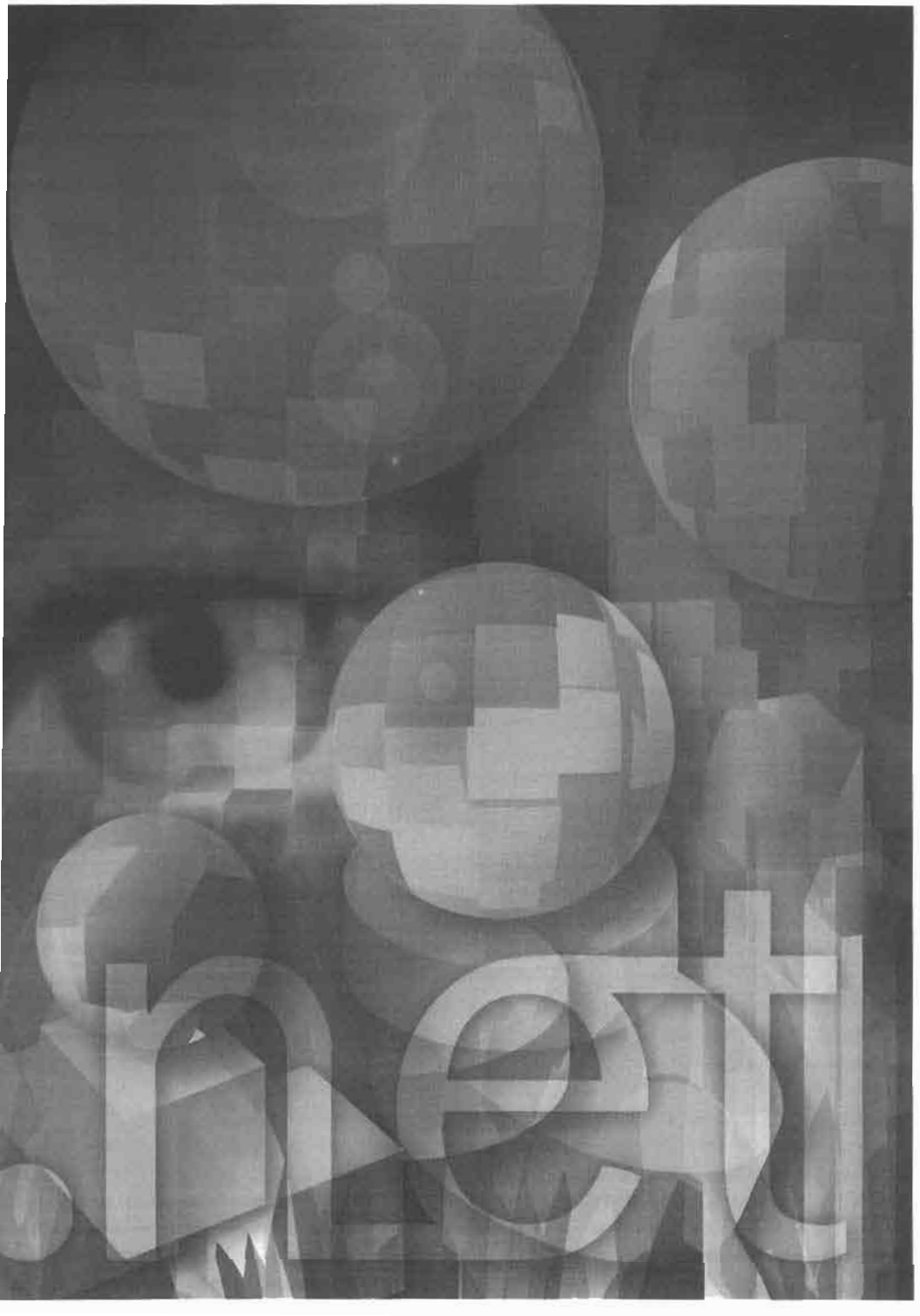
Figura 14.7. Visualización previa en una ventana propia

Resumen

Este capítulo nos ha servido, principalmente, para familiarizarnos con el mecanismo de publicación de información con que cuenta Visual Basic .NET, representado por componentes como `PrintDocument`, `PrintPreviewDialog` y `PrintPreviewControl`. Ya que se puede seleccionar cualquier dispositivo

de salida, es posible no sólo imprimir los datos sino también enviarlos por fax o a un archivo para su posterior envío o conversión.

La tarea de composición del documento se asemeja bastante a la descrita en el capítulo previo para dibujar sobre un formulario, ya que cada página que va a imprimirse aparece a nuestros ojos como un objeto `Graphics`. Esto nos permite utilizar exactamente los mismos elementos descritos en ese capítulo: brochas, lápices, tipos de letra, etc.


next

15 Desarrollo de componentes

En todos los ejemplos que hemos desarrollado hasta ahora, desde los primeros capítulos, hemos estado utilizando multitud de clases para, bien directamente o bien tras la creación de un objeto, acceder a servicios concretos de la plataforma .NET.

En los últimos capítulos dichos objetos los hemos creado de manera visual, arrastrándolos sobre un contenedor y configurando su funcionamiento mediante la edición de propiedades. Cuando un objeto puede manipularse de esta forma, mediante un diseñador, hablamos de componentes en lugar de llamarlos simplemente objetos. Un componente, por lo tanto, es un objeto creado a partir de una clase, como cualquier otro, pero que dispone de elementos que facilitan su tratamiento visual.

A pesar del gran abanico de componentes de todo tipo existentes en la plataforma .NET, está clara la imposibilidad de cubrir las necesidades de todos los desarrolladores. Por eso existen las terceras empresas dedicadas a la creación y venta de componentes para las distintas herramientas, no siendo la plataforma .NET una excepción. Nosotros, como desarrolladores, también podemos crear componentes, ya sea para nuestros propios proyectos o para proveerlos a terceros.

El objetivo de este capítulo es iniciarle en el desarrollo de componentes .NET, clases que pueden utilizarse genéricamente y de manera indistinta en una aplicación Windows, una aplicación Web u otro tipo de proyecto. Posteriormente, en los dos capítulos siguientes, entraremos en los detalles específicos de la creación de controles para formularios Windows, controles ASP.NET, etc.

Conceptos básicos

Antes de entrar en los detalles del desarrollo de componentes no está de más que establezcamos los conceptos básicos que deberemos conocer, así como la terminología que va a utilizarse en el resto del capítulo.

Entre otros elementos, Visual Basic .NET cuenta con un diseñador para formularios Windows y otro para formularios Web, que hemos conocido en capítulos previos, de tal manera que el diseño de interfaces de usuario Windows y Web se lleva a cabo con operaciones de arrastrar y soltar. Estos diseñadores permiten seleccionar los componentes, alojarlos en un contenedor, personalizarlos editando sus propiedades, asociar código a los eventos, etc.

Para desempeñar su función, un cierto diseñador actuará básicamente sobre dos elementos: un contenedor y uno o más componentes. El contenedor es también un componente, aunque con la particularidad de poder alojar otros en su interior. Un ejemplo de contenedor es un formulario Windows o bien una página Web.

Una clase escrita en Visual Basic .NET puede ser alojada en una biblioteca y utilizada desde otra aplicación directamente, como si de un componente se tratase. Esa clase, sin embargo, no podría añadirse a la paleta de componentes de un diseñador a menos que esté derivada de `System.ComponentModel.Component` o implemente la interfaz `IComponent` de ese mismo ámbito con nombre.

Análogamente, una clase no podrá considerarse un control a menos que esté derivada, directa o indirectamente, de `System.Windows.Forms.Control`, en el caso de controles para formularios Windows, o `System.Web.UI.Control`, para los controles ASP.NET.

Para resumir este punto en una frase, haciendo referencia a los distintos conceptos comentados, podríamos decir que los *componentes* y *controles* pueden ser alojados en un *contenedor* y personalizados mediante un *diseñador*.

¿Cuándo un objeto es un componente?

Ya sabemos que para que un objeto pueda ser considerado un componente debe poder manipularse visualmente mediante un diseñador pero, ¿qué diferencia existe en una clase de componente para contar con esa posibilidad? Existen dos opciones: o dicha clase implementa la interfaz `IComponent` directamente o hereda esa implementación de la clase `Component`. Ambos elementos, la interfaz `IComponent` y la clase `Component`, se encuentran definidos en el ámbito `System.ComponentModel`. Otro requerimiento indispensable es que la clase disponga de un constructor que no necesite parámetro alguno ya que, de lo contrario, no podrían crearse objetos visualmente, arrastrándolos sobre un contenedor.

La interfaz `IComponent` establece las propiedades y métodos indispensables para que una clase pueda ser considerada como molde para crear componentes. En realidad los miembros son tan sólo dos: la propiedad `Site` y el

método `Disposed`. La primera facilitará el acceso a la interfaz `ISite` del contenedor donde se inserte el componente, medio por el cual se establecerá la comunicación entre componente y contenedor. El método `Disposed` es heredado por la interfaz `IComponent` de la interfaz `IDisposed`, de la cual está derivada. Dicho método se encargará de liberar los recursos cuando el componente sea destruido.

Aunque podríamos implementar manualmente la interfaz `IComponent` en cualquier clase propia, a fin de convertirla en una clase de componente, resulta mucho más fácil derivar, directa o indirectamente, de la clase `Component`. Ésta es la base genérica de la mayoría de los componentes que hemos utilizado en capítulos previos, especialmente al trabajar con formularios Windows. `Component` se encarga de implementar los miembros de `IComponent` y, además, nos ofrece algunas propiedades y métodos protegidos, que podemos usar al derivar de ella.

En la práctica, por lo tanto, para crear una nueva clase de componente lo que haremos será tomar como base la clase `Component` o, en su defecto, otra clase que, directa o indirectamente, esté derivada de `Component`. Con esto podremos centrarnos en el desarrollo de la funcionalidad propia del componente, sin preocuparnos demasiado por los detalles de implementación genérica de cualquier componente.

Nota

Al definir una clase de componente hemos de tener en cuenta que, aunque puede crearse y ser utilizado mediante código como cualquier otra clase, el objetivo es facilitar la manipulación en un entorno visual. Con ese fin es fundamental que la clase defina mediante propiedades y eventos todos los atributos con que puedan contar los objetos, en vez de recurrir a métodos para efectuar esa configuración. Debe tener en cuenta que los diseñadores son capaces de mostrar las propiedades con que cuenta un componente y facilitar su edición, pero no existe la posibilidad de invocación a métodos durante el diseño.

Un sencillo ejemplo

Antes de continuar con las explicaciones teóricas relativas al diseño de clases de componentes, y puesto que ya sabemos cuál es la base fundamental para su construcción, vamos a desarrollar un sencillo ejemplo. La finalidad del primer componente que vamos a diseñar será facilitar la conversión de números entre la base dos y la base diez, es decir, convertir números decimales en binarios o viceversa.

Para conseguir esto el componente no precisa interfaz de usuario, sino simplemente de las propiedades adecuadas para que sea posible asignar un valor y obtener la conversión.

El componente, al que llamaremos `Convertidor`, dispondrá de dos propiedades. Al asignar un valor a `Binario`, concretamente una cadena con una secuencia de unos y ceros, lo guardaremos en una variable interna y, además, procesaremos el parámetro para obtener el número decimal correspondiente. Éste podrá obtenerse de la propiedad `Valor`.

De manera análoga, una asignación a esta última propiedad efectuará el proceso inverso.

Inicie un nuevo proyecto de tipo **Biblioteca de clases** (véase figura 15.1), obteniendo así un módulo donde definir una clase. Puede insertar el código mostrado más abajo directamente en ese módulo o, si lo prefiere, utilizar la ventana **Agregar nuevo elemento** para añadir al proyecto una clase de componentes, como se ha hecho en la figura 15.2.

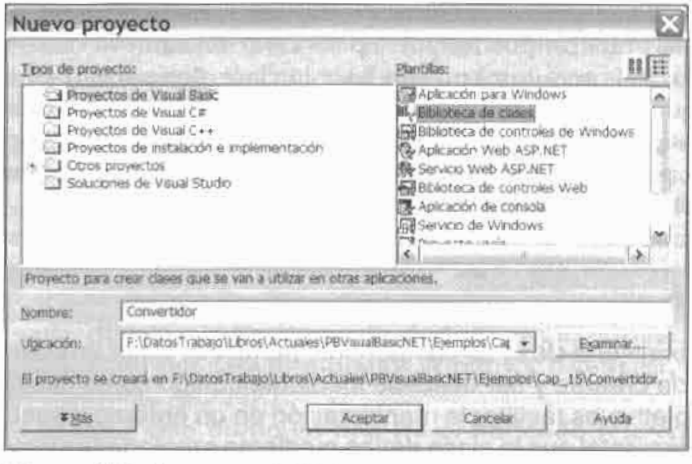


Figura 15.1. Los componentes deben alojarse en bibliotecas de clases

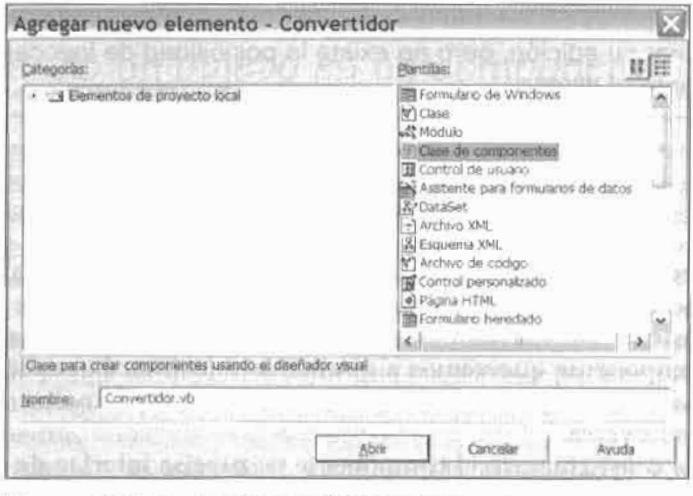


Figura 15.2. Podemos añadir al proyecto un esqueleto de componente prefabricado

Todo el código central del componente es el que puede ver a continuación. El método `Get` de cada propiedad es realmente sencillo, limitándose a devolver el valor que contienen las variables `Fbinario` y `Fvalor`. Por defecto, inicialmente, dichas variables contienen la cadena `"0"` y el valor entero `0`.

```
Public Class Convertidor
  Inherits Component

  Private Fbinario As String = "0"
  Private Fvalor As Integer = 0

  Public Property Binario() As String

    Get
      Return Fbinario
    End Get

    Set(ByVal Value As String)

      Fbinario = Value
      Fvalor = 0

      Dim Contador As Integer
      Dim Exp As Integer = 0
      For Contador = Binario.Length To 1 Step -1

        Fvalor += System.Math.Pow(2, Exp) * _
           Char.GetNumericValue(Fbinario.Chars(Contador))
        Exp += 1
      Next
    End Set
  End Property

  Public Property Valor() As Integer

    Get
      Return Fvalor
    End Get

  Set(ByVal Value As Integer)
      Fvalor = Value
      Fbinario = ""

      Do While Value >= 2
```

```
            Fbinario = IIf(Value Mod 2 = 0, "0", "1") _
              & Fbinario
            Value \= 2
          Loop
            [illegible]
            [illegible]
          Fbinario = Value.ToString() & Binario
        End Set
      End Property
    End Class
```

Como puede ver, prácticamente toda la lógica del componente se encuentra alojada en los métodos de asignación de las propiedades. La conversión de decimal a binario es una simple división sucesiva entre dos, formando el número binario con todos los restos de esas divisiones y el último cociente. El proceso inverso, la conversión de binario a decimal, se efectúa elevando la base, que es 2, al exponente que corresponda a la posición de cada dígito. Para este fin se utiliza el método `Pow()` de la clase `Math`.

Nota

La propiedad `Chars` del tipo `String` permite acceder individualmente a cada uno de los caracteres de una cadena, utilizando un índice comprendido entre 0 y el número de caracteres existentes menos 1. El método `GetNumericValue()` de la clase `Char`, como puede suponer, obtiene el valor numérico del carácter entregado como parámetro, convirtiendo el `Char` en un `Double` sobre el que podemos operar matemáticamente.

Finalizado el desarrollo, el siguiente paso será compilar el proyecto con el fin de obtener un ensamblado, en este caso una biblioteca de enlace dinámico, con el código IL del componente. Esa biblioteca podemos alojarla en un repositorio común donde tengamos todos nuestros componentes o los componentes de terceros.

Instalación del componente

Indistintamente del tipo de aplicación que hayamos iniciado, para utilizar nuestro componente `Convertidor` desde Visual Basic .NET lo primero que tendremos que hacer será instalarlo en el Cuadro de herramientas. Con este fin pulsamos sobre él con el botón secundario del ratón, desplegando el menú asociado, y elegimos la opción Personalizar cuadro de herramientas. Aparece un cuadro de diálogo en el que existen dos páginas. Abrimos la titulada **Componentes de .NET Framework** y pulsamos el botón **Examinar** para localizar la biblioteca donde se encuentra el ensamblado del componente.

A partir de este momento el componente ya aparecerá, como puede verse en la figura 15.3, en la lista de componentes que podemos agregar al Cuadro de herramientas. Lo seleccionamos y cerramos el cuadro de diálogo pulsando el

botón **Aceptar**. Normalmente el componente aparecerá en la página que teníamos abierta, al final de la lista de componentes que existiese en ese momento (véase figura 15.4). No tenemos más que pinchar sobre él e insertarlo donde nos convenga.

Figura 15.3. Añadimos el componente a la lista de componentes .NET

Figura 15.4. El componente ya en el Cuadro de herramientas

Este procedimiento será el habitual siempre que pretendamos usar el componente de forma visual, como vamos a ver en el punto siguiente. Si tan sólo lo necesitamos para crearlo desde el código de nuestro programa, bastaría con añadir una referencia al proyecto utilizando la opción **Agregar referencia** del menú asociado a la carpeta **Referencias**, en la ventana del **Explorador de soluciones**. También podemos necesitar la inserción de una sentencia `Imports` al inicio del código para importar el ámbito en que se encuentra el componente.

Uso visual del componente

Habiendo iniciado un proyecto, ya sea basado en formularios Windows o formularios Web, y añadido nuestro nuevo componente al Cuadro de herramientas, su uso es prácticamente idéntico al de cualquier otro componente que hayamos podido usar en los diversos ejemplos que se han desarrollado hasta ahora. Haciendo doble clic sobre el componente, o bien arrastrándolo hasta el contenedor, insertaremos una copia. En ese momento habremos creado un objeto a partir de la clase `Convertidor`, un componente.

Observe que en la ventana Propiedades aparecen las propiedades que habíamos definido en nuestra clase. En la figura 15.5 puede ver cómo hemos alterado la propiedad `Valor` y obtenido su equivalente en `Binario`. Lógicamente también podemos efectuar la operación inversa, modificando `Binario` para obtener el resultado en `Valor`. Dado que el componente no tiene interfaz de usuario, podríamos introducir en el formulario un par de cajas de texto para habilitar la conversión en ejecución. El código se limitaría a una simple asignación a `Valor` y lectura de `Binario` o viceversa.

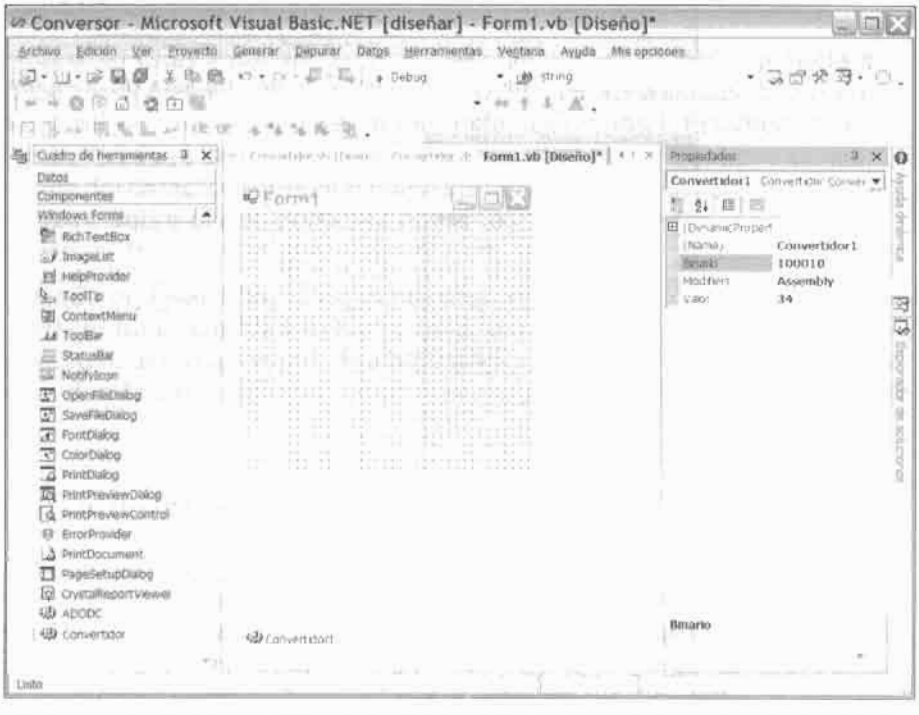


Figura 15.5. Podemos utilizar el componente `Convertidor` de forma visual

A pesar de que el componente `Convertidor` aparece en el Cuadro de herramientas y podemos crearlo y manipularlo mediante un diseñador, lo cierto es que su comportamiento como componente aún deja que desear. Al modificar en la ventana Propiedades el valor de la propiedad `Valor` la conversión no

aparece automáticamente en `Binario`, siendo necesario desplazarse hasta dicha propiedad para ver el resultado. Lo mismo ocurre en sentido inverso. Otro comportamiento anómalo se produce cuando asignamos un valor a `Binario`. En teoría el número debería estar compuesto sólo de 0 y 1, puesto que es binario. En la práctica, sin embargo, puede comprobar que es posible la introducción de cualquier otro dígito.

Mejoras para la fase de diseño

Al crear un componente hay que pensar que el usuario lo usará en el interior de una herramienta de diseño, como pueda ser Visual Basic .NET. Aparte de implementar la funcionalidad propia del componente, como son los métodos de conversión en uno de nuestros ejemplos previos, también podemos incluir en la misma biblioteca otros elementos, como atributos, convertidores de valores, editores de propiedades, diseñadores específicos, etc.

Muchos de estos elementos se definen mediante atributos que, como sabe, se colocan delante de distintos elementos del lenguaje delimitados entre los caracteres < y >.

Los atributos se incluyen en el ensamblado como meta-información, datos que describen aspectos concretos de los objetos o sus miembros pero que no se implementan mediante código.

Atributos para componentes

Si ha observado la ventana **Propiedades**, habrá podido ver que en ella las distintas propiedades están clasificadas en categorías. Esto simplifica el trabajo, de manera que si estamos definiendo la apariencia de un control no tenemos que estar buscando propiedades dispersas al estar ordenadas alfabéticamente. Al seleccionar una propiedad cualquiera habrá visto que en la parte inferior aparece una descripción. La categoría de una propiedad, su descripción, la propiedad por defecto de una clase y otros elementos similares se definen mediante atributos.

Inserte nuestro control de conversión en un formulario y fíjese en lo que ocurre. Las dos propiedades que hemos definido aparecen al final de la ventana de propiedades, en una categoría llamada **Varios**. Al seleccionar cualquiera de ellas, en la parte inferior tan sólo aparece el nombre y no una descripción. Esto ocurre porque en nuestra clase no hemos incluido los atributos necesarios para facilitar toda esta información.

Podemos asociar una lista de atributos a una clase, una propiedad, un método, un evento y, en general, casi a cualquier miembro que pueda contener la clase. Los atributos van contenidos, como se ha dicho antes, entre los símbolos < y > y, en caso de existir varios, se separan con comas. Cada atributo se compone de una palabra clave y un valor facilitado entre paréntesis. La tabla 15.1 enumera algunos de los atributos de uso más corriente.

Tabla 15.1. Atributos en controles

Atributo	Descripción
`Bindable`	Indica si la propiedad puede enlazarse con un origen de datos
`Browsable`	Indica si la propiedad debe aparecer o no en la ventana de propiedades
`Category`	Asocia la propiedad a una determinada categoría
`DefaultEvent`	Establece el evento por defecto para el control
`DefaultProperty`	Establece la propiedad por defecto para el control
`DefaultValue`	Fija el valor por defecto que tendrá la propiedad
`Description`	Facilita la descripción para la propiedad o evento
`Editor`	Asocia un editor específico con una propiedad
`TypeConverter`	Asocia un conversor de tipo con una propiedad
`Designer`	Asocia un diseñador específico con el componente

Utilizando algunos de estos atributos podríamos modificar nuestro control dejándolo como se muestra en el código siguiente, en el cual se ha omitido el código de implementación de las propiedades puesto que éste no sufre ningún cambio.

```
<DefaultProperty("Valor")> _
Public Class Convertidor
  Inherits System.ComponentModel.Component
   ...
  Private Fbinario As String = "0"
  Private Fvalor As Integer = 0

  <Category("Apariencia"), _
   Description("Valor representado en binario"), _
   DefaultValue("00000000")> _
  Public Property Binario() As String
    ...
  End Property

  <Category("Apariencia"), _
   Description("Contiene el valor en decimal"), _
   DefaultValue(0) > _
  Public Property Valor() As Integer
      ...
  End Property
End Class
```

Si tras introducir estos cambios compila el componente y lo utiliza podrá advertir varios cambios. Ahora las dos propiedades ya no aparecen en la sección de misceláneas sino al principio, en la categoría Apariencia. Al seleccionar cualquiera de ellas verá una descripción en la parte inferior. Observe, además,

que se indica cuál es el valor por defecto de esta propiedad. En la ventana **Propiedades** el valor aparecerá destacado en negrita si se ha modificado o, por el contrario, con el tipo normal si se mantiene el valor por defecto. Al insertar un nuevo componente `Convertidor`, o al cambiar de un componente a otro la selección, verá que la propiedad que aparece seleccionada por defecto en la ventana **Propiedades** es `Valor`, ya que así lo hemos indicado mediante el atributo `DefaultProperty` de la clase. En la figura 15.6 puede ver el control con la propiedad `Valor` seleccionada y su descripción en la parte inferior.

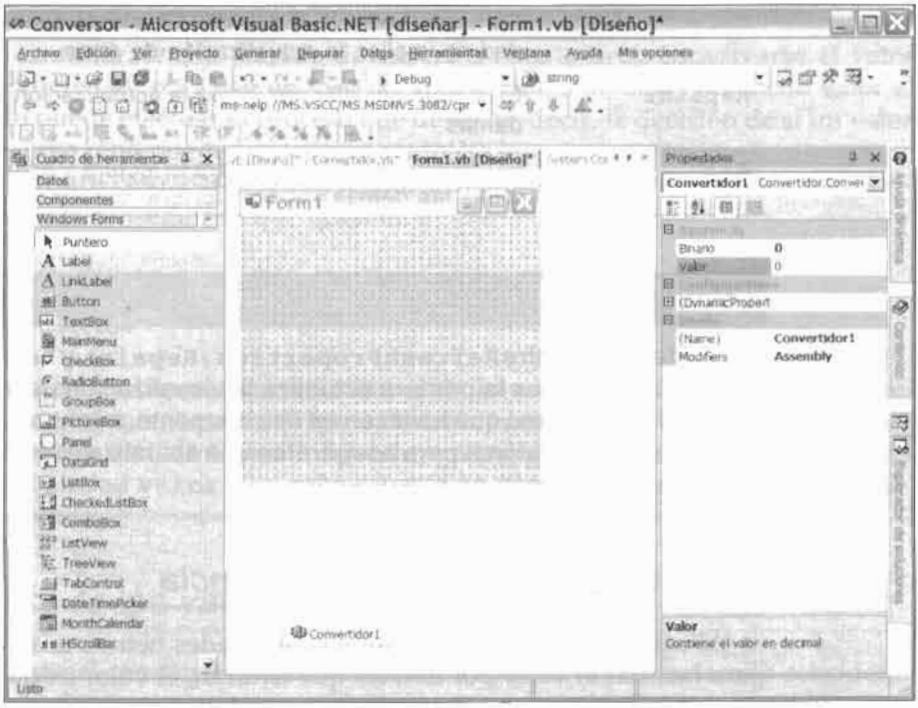


Figura 15.6. Los atributos asociados al control son usados por el entorno de diseño

Actualización automática de la ventana Propiedades

Uno de los problemas que plantea nuestro componente, tal y como está diseñado en este momento, es que al modificar la propiedad `Valor` la actualización de `Binario` no es inmediata, ocurriendo lo mismo en sentido inverso. Por regla general la modificación de una propiedad de un componente no implica cambios en otra, por ello la actualización de los valores no se produce de manera automática, como sería de desear en este caso.

Este comportamiento por defecto, al igual que muchos otros factores del comportamiento en la fase de diseño, se controla mediante atributos. En este caso el atributo que nos interesa es `RefreshProperties`, que puede tomar como valor uno de los valores que se enumeran en la tabla 15.2. Añadiendo

`RefreshProperties(RefreshProperties.All)` a la lista de atributos de las propiedades `Binario` y `Valor`, y tras compilar de nuevo la biblioteca que contiene el componente, podrá ver que ahora el comportamiento es mucho más lógico. Al cambiar el valor de una de las propiedades la otra se actualiza de forma inmediata, mostrando la conversión.

Tabla 15.2. Valores para el atributo `RefreshProperties`

Valor	Indica
`Default`	Comportamiento por defecto. El cambio de valor de esta propiedad no implica la actualización de otras
`Repaint`	El cambio de valor provocará la actualización visual de todas las demás
`All`	El cambio de valor provocará la lectura y actualización visual de todas las demás

Nota

La diferencia entre `RefreshProperties.Repaint` y `RefreshProperties.All` es que la primera actualiza la visualización de las propiedades usando los valores que tuviesen en ese momento, mientras que la segunda releerá dichos valores para asegurarse una actualización completa, que es lo que nos interesa en nuestro caso.

Valores por defecto y persistencia

Al asignar atributos a nuestras propiedades hemos usado `DefaultValue` para facilitar el valor por defecto que tendrán, el valor inicial. También hemos visto que en la ventana Propiedades ese valor aparece con el tipo normal, mientras que se muestra en negrita cuando ha sido modificado. En realidad dicho atributo no tiene una finalidad visual, sino que determina la persistencia del valor de una propiedad.

Puede comprobar lo que ocurre al modificar el valor de una propiedad de `Convertidor` examinando el código que genera el diseñador. Incluya el componente en un formulario Windows, por ejemplo, y compruebe el código. Verá que no se asigna explícitamente ningún valor a `Convertidor1` ya que `Valor` y `Binario` mantienen su valor por defecto. Modifique el contenido de `Valor` y observe el código. Ahora aparecen sendas sentencias de asignación, una para `Valor` y otra para `Binario`. De esta manera, al ejecutar el programa, el componente tendrá en principio los valores que se establecieron durante la fase de diseño.

El atributo `DefaultValue` es uno de los mecanismos con que cuenta Visual Basic .NET para controlar la persistencia, mediante código, del valor de las propiedades, pero no el único. En el caso de nuestro componente la asigna-

ción de valor a una de las propiedades implica automáticamente el cálculo del valor de la otra, por lo que no tiene mucho sentido que el diseñador conserve la asignación de los valores establecidos en la fase de diseño como dos sentencias independientes. Con asignar a `Valor` sería suficiente para que `Binario` tuviese su valor. `Binario`, por tanto, no tendría por qué conservarse en el código de iniciación del proyecto.

Existe una opción, para controlar la persistencia de las propiedades, que es mucho más flexible que el atributo `DefaultValue`. Consiste en añadir a la clase un método privado que tenga por nombre `ShouldSerializeXXXXX() As Boolean`, donde `XXXXX` sería el nombre de la propiedad cuya persistencia se desea controlar. Dicho método devolverá `True`, si debe conservarse el valor de la propiedad, o `False` en caso contrario. Antes de devolver dicho valor el método puede efectuar el proceso que desee, es decir, la decisión de si un valor se conserva o no puede ser todo lo compleja que se necesite.

En nuestro caso, bastaría con añadir el método siguiente a la clase `Convertidor` para conseguir lo que queremos:

```
Private Function ShouldSerializeBinario() As Boolean
  Return False
End Function
```

También tendríamos que eliminar el atributo `DefaultValue` de la propiedad `Binario` que, a partir de ese momento, nunca conservará su valor de manera explícita, indistintamente del valor que tenga. Al conservarse el contenido de la propiedad `Valor`, sin embargo, `Binario` también tendría siempre el dato apropiado.

Validez de los valores asignados

Aunque el comportamiento de nuestro componente ha mejorado considerablemente desde su primera versión, aún siguen existiendo algunos puntos que pueden mejorarse. Pruebe a asignar a `Binario` un número que no sólo contenga los dígitos 0 y 1, incluso conteniendo caracteres alfabéticos. Verá que lo permite. Pruebe también a asignar a `Valor` cualquier número negativo y verá lo que ocurre. Lógicamente hay problemas que resolver en la implementación de `Convertidor`.

Cuando se define una propiedad el método `Set` no está ahí sólo para guardar el valor facilitado y, como en nuestro caso, efectuar algún proceso para obtener un resultado. Ese método también debe utilizarse para comprobar que el valor entregado es válido, evitando en caso contrario su conservación en los métodos internos de la clase. Lo habitual, para comunicar el problema al diseñador o al código que está usando el componente, es generar una excepción.

Deberíamos, por tanto, modificar los métodos `Set` de nuestras propiedades para efectuar las comprobaciones necesarias, generando una excepción en caso de que el valor facilitado sea inválido. De esta forma, al asignar un valor erróneo utilizando la ventana Propiedades aparecerá una ventana similar a la de la

figura 15.7, comunicando el error, y el valor aparecerá seleccionado en la propia ventana **Propiedades**. Podemos modificarlo, introduciendo un valor correcto, o bien pulsar la tecla **Esc** para devolverle el valor que tenía anteriormente.

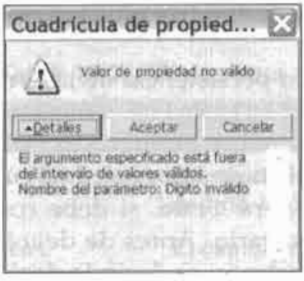


Figura 15.7. Mensaje que indica la asignación de un valor erróneo a la propiedad `Binario`

Eventos y atributos

Un componente, como cualquier otra clase, puede generar eventos. Es más, en el caso de las clases de componentes los eventos son un medio imprescindible para comunicarse con la aplicación en que vayan a ser utilizados. Ya sabe cómo definir sus propios eventos, así como generarlos y gestionarlos. En los componentes la mayoría de los eventos son de tipo `EventHandler`, un tipo que establece que el método que actúe como gestor recibirá dos parámetros: una referencia al componente que ha generado el evento y un objeto `EventArgs` con los parámetros que sean aplicables.

Al igual que las propiedades y las clases, los eventos pueden tener asociados atributos que, por ejemplo, establezcan su categoría o faciliten una descripción. Al trabajar con Visual Basic .NET esos elementos no son accesibles, pero si trabajásemos con Visual C#, por ejemplo, veríamos que la lista **Propiedades** también muestra una lista de eventos, apareciendo éstos agrupados por categorías y con una descripción. En la figura 15.8 puede ver los dos eventos añadidos al componente `Convertidor` en la ventana **Propiedades** tras haber iniciado un proyecto con C#.

A continuación tiene el código completo del componente `Convertidor` en su versión final, con la definición y generación de los eventos y cambios en los métodos `Set` de las propiedades para verificar el valor y, en caso de encontrar algún error, generar la correspondiente excepción.

```
Imports System.ComponentModel


<DefaultProperty("Valor"), DefaultEvent("ValorChanged")> _
Public Class Convertidor
  Inherits System.ComponentModel.Component
```

```
Private Fbinario As String = "0"
Private Fvalor As Integer = 0

<Category("PropertyChanged"), _
 Description("Comunica el cambio de valor en la propiedad")> _
Public Event BinarioChanged As EventHandler

<Category("PropertyChanged"), _
 Description("Comunica el cambio de valor en la propiedad")> _
Public Event ValorChanged As EventHandler

<Category("Apariencia"), _
 Description("Valor representado en binario"), _
 RefreshProperties(RefreshProperties.All)> _
Public Property Binario() As String

  Get
    Return Fbinario
  End Get

  Set(ByVal Value As String)

    If Value = Fbinario Then Return

    Dim Resultado As Integer

    Dim Contador As Integer
    Dim Exp As Integer = 0
    For Contador = Value.Length - 1 To 0 Step -1

      If "01".IndexOf(Value.Chars(Contador)) = -1 Then
        Throw New ArgumentOutOfRangeException("Dígito inválido")
      End If

      Resultado += System.Math.Pow(2, Exp) * _
        Char.GetNumericValue(Value.Chars(Contador))
      Exp += 1
    Next

    Fbinario = Value
    Fvalor = Resultado

    RaiseEvent BinarioChanged(Me, EventArgs.Empty)
  End Set
End Property

Private Function ShouldSerializeBinario() As Boolean
  Return False
End Function
```

```
  <Category("Apariencia"), _
   Description("Contiene el valor en decimal"), _
   DefaultValue(0), RefreshProperties(RefreshProperties.All)> _
   Public Property Valor() As Integer

    Get
      Return Fvalor
    End Get

    Set(ByVal Value As Integer)

      If Value = Fvalor Then Return

      If Value < 0 Then Throw _
          New ArgumentOutOfRangeException("Valor inválido")

      Fvalor = Value
      Fbinario = ""

      Do While Value >= 2
        Fbinario = IIf(Value Mod 2 = 0, "0", "1") _
          & Fbinario
        Value \= 2
      Loop

      Fbinario = Value.ToString() & Binario

      RaiseEvent ValorChanged(Me, EventArgs.Empty)
    End Set
  End Property
End Class
```

Figura 15.8. Los eventos también se agrupan en categorías

El método `Set` que más cambios ha sufrido es el de la propiedad `Binario`, ya que se ha trasladado la conservación de los nuevos valores al final del método, tras verificar que no se encuentra ningún dígito inválido en la cadena entregada como parámetro.

Fíjese en cómo usamos el método `IndexOf()` de la clase `String` para comprobar que cada carácter de ese parámetro sólo sea 0 ó 1.

Nota

Observe que al inicio de la clase se ha definido un evento por defecto. Ese evento será el que se va a utilizar al hacer doble clic sobre un componente `Binario`.

Editores y convertidores a medida

En los ejemplos de controles de este capítulo siempre hemos definido propiedades de tipos básicos, como `String` o `Integer`. Los editores de propiedades de Visual Basic .NET saben cómo convertir esos valores en una representación apropiada para el usuario, permitiéndole así ver y modificar el contenido de la propiedad. En ocasiones, sin embargo, puede darse el caso de que el tipo de una propiedad no sea uno de los fundamentales y, por tanto, no sea posible editarlo directamente. En estos casos existen varias alternativas.

Una de las opciones posibles es crear un objeto convertidor que, mediante ciertos métodos clave, se encargue de la conversión del valor de la propiedad a un tipo representable en la ventana de propiedades. La clase de dicho objeto deberá derivarse de `TypeConverter`, que es una clase definida en el ámbito `System.ComponentModel` y de la que es necesario redefinir básicamente tres métodos: `CanConvertFrom()`, `ConvertFrom()` y `ConvertTo()`. El primero indicará si es posible o no convertir un cierto tipo al tipo de la propiedad, mientras que los otros dos efectuarán la conversión en ambos sentidos.

Si un convertidor de tipo no es suficiente, siempre cabe la posibilidad de crear un editor de propiedades específico para nuestra propiedad. Esta es una tarea relativamente más compleja y, además, hay que tener en cuenta que un editor de propiedades establece una estrecha conexión con el diseñador en el que se utiliza, de tal forma que hay una dependencia entre ambos y el editor no podría ser usado en otro diseñador.

Para crear un editor de propiedades personalizado deberíamos derivar una nueva clase tomando como base a `System.Drawing.Design.UITypeEditor`, redefiniendo los métodos `GetEditStyle()`, para establecer el tipo del editor, y `EditValue()`, que pondría en marcha la edición del valor de la propiedad.

Los convertidores y editores específicos, junto con los atributos previamente mencionados, son los elementos más habituales que podremos añadir para mejorar el uso de nuestros componentes en las herramientas de diseño. Podemos, no obstante, crear otras utilidades como extensores, que añaden propiedades a otros controles e, incluso, nuestros propios diseñadores.

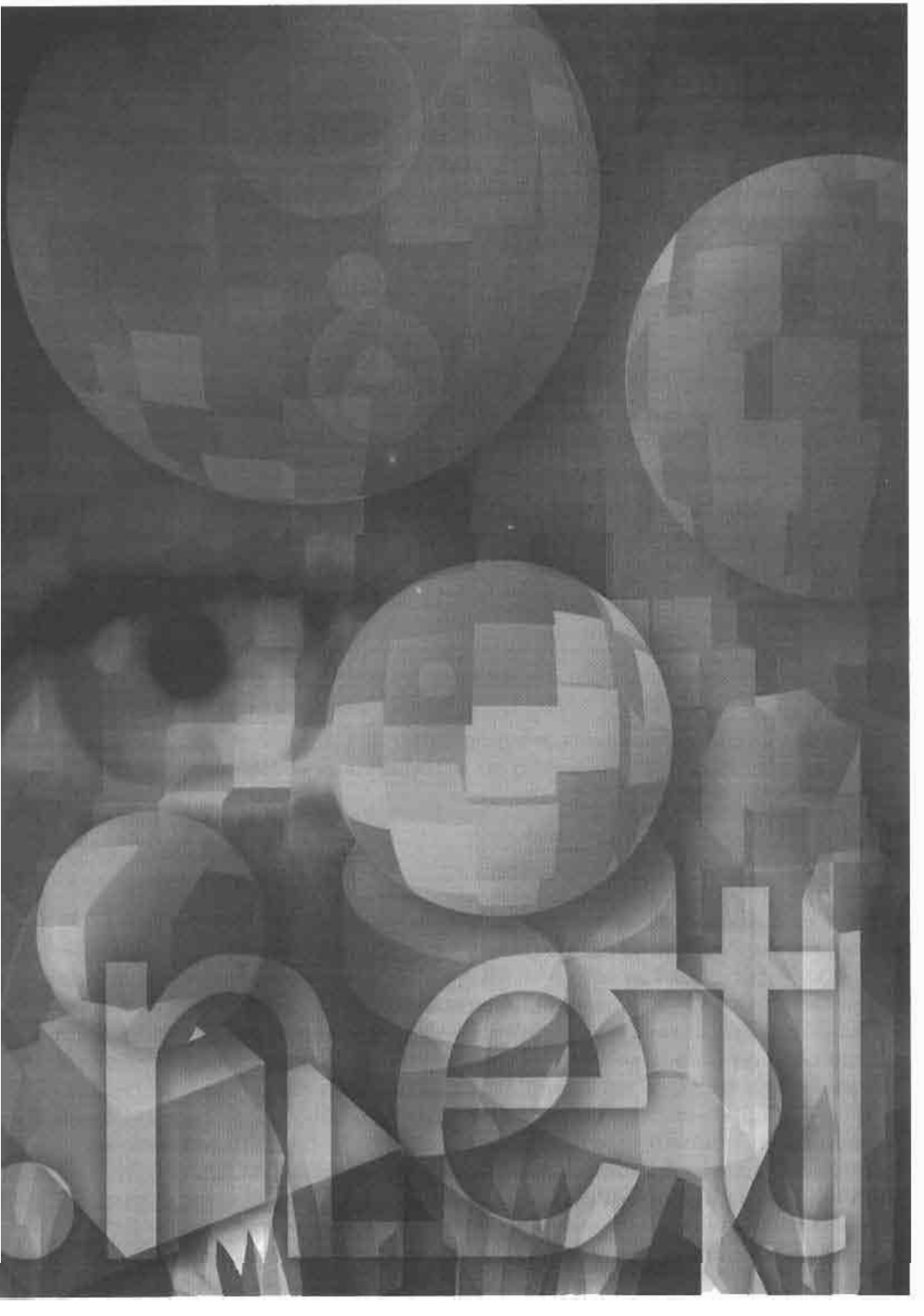
Puntos clave

- Los componentes son objetos construidos para poder alojarse en el interior de un contenedor y, eventualmente, manipularse mediante un diseñador.
- Para que una clase pueda considerarse una clase de componente debe implementar la interfaz `IComponent`, para lo cual basta con derivar la nueva clase tomando como base `Component`.
- Al desarrollar un componente hay que implementar la mayor parte posible de su funcionalidad mediante propiedades y eventos, contribuyendo a su personalización en la fase de diseño en lugar de escribiendo código.
- Mediante atributos es posible mejorar el comportamiento de los componentes en el diseñador, agrupando propiedades y eventos en categorías, facilitando descripciones, valores por defecto, etc.
- La persistencia de las propiedades de un componente puede controlarse mediante el atributo `DefaultValue` o bien implementando un método privado `ShouldSerialize()` que decida cuándo debe o no mantenerse el valor de la propiedad.
- Al construir una clase de componente hay que aprovechar los métodos `Set` de las propiedades para asegurar que los valores sean válidos.
- Es posible facilitar convertidores propios, editores y diseñadores específicos para las propiedades de nuestros componentes.
- Una vez compilados, los componentes se alojan en una biblioteca que podemos usar desde cualquier tipo de proyecto.

Resumen

Como ha podido ver en este capítulo, el desarrollo de componentes .NET es poco más que una extensión de la definición de clases de objetos que ya conocíamos. Una clase de componente dispone de métodos, propiedades y eventos, como cualquier otra, si bien se hace mayor hincapié en las posibilidades de manipulación durante la fase de diseño. Por ello se hace un uso intensivo de los atributos a fin de facilitar información descriptiva al entorno, información con la cual el diseñador puede agrupar los miembros o decidir si un cierto valor en una propiedad debe o no escribirse en el módulo de código.

La clase `Component`, que hemos usado como base de nuestro componente de ejemplo, facilita una implementación estándar de la interfaz `IComponent`. De manera análoga, existen clases que pueden servirnos como base para construir no ya componentes genéricos, como `Convertidor`, sino controles específicos para aplicaciones Windows o aplicaciones Web. Será el tema que vamos a abordar en los dos próximos capítulos.


.net

16 Controles Windows

Los componentes .NET, derivados de `System.ComponentModel.Component`, pueden utilizarse genéricamente en cualquier tipo de proyecto que podamos desarrollar con Visual Basic .NET pero, a cambio, no aprovechan las posibilidades que tiene a su alcance un control específico para formularios Windows, por poner un ejemplo.

Como ya se apuntaba al final del capítulo previo, en el que conocimos los fundamentos de desarrollo de componentes, en éste vamos a conocer los elementos necesarios para crear nuestros propios controles Windows, entendiendo como tales aquellos que están diseñados específicamente para ser insertados en un formulario Windows. Conocerá las distintas clases que puede utilizar como bases de sus propios controles y aplicará este conocimiento en la práctica con varios ejemplos.

Toda la información del capítulo previo resultará fundamental para poder abordar éste. No en vano, un control Windows no es más que un componente .NET con una serie de características concretas. La forma de implementar propiedades, eventos y métodos, describir su funcionamiento mediante atributos, etc., será, por tanto, idéntica a la explicada en dicho capítulo.

Tipos de controles Windows

Al crear una nueva clase de control tenemos varias opciones en cuanto a la clase que actuará como base: `Control`, `ScrollableControl`, `UserControl`

y `ContainerControl`, todas ellas alojadas en el ámbito `System.Windows.Forms`. En realidad las posibilidades son muchas más, ya que podemos tomar como base la clase de cualquier control existente. Estas cuatro clases podríamos decir que son las raíces fundamentales del resto.

En la parte más alta del árbol jerárquico se encuentra la clase `Control`, derivada de `System.ComponentModel.Component`, clase que ya conocemos. La clase `Control`, por tanto, ya dispone de una implementación por defecto de `IComponent`, característica indispensable para todo componente .NET indistintamente de su tipo.

Además, la clase `Control` se ocupa de tareas comunes a todos los controles Windows: posición y dimensiones, gestión de la entrada de datos por teclado y ratón, obtención de un manejador de ventana, etc. `Control` es la base adecuada para crear controles totalmente nuevos, no con una funcionalidad similar a uno ya existente, y que se ocupan de facilitar su apariencia para mostrarse en el interior del formulario.

Una de las clases derivadas de `Control` es `ScrollableControl`, una clase que extiende la funcionalidad de `Control` para facilitar la visualización de barras de desplazamiento, indispensables cuando el contenido que debe mostrar el control excede de sus dimensiones. Las propiedades y métodos añadidos por `ScrollableControl` controlan la visualización de las barras y el desplazamiento, automático o manual, del contenido del control.

Si necesitamos que nuestro nuevo control pueda, a su vez, contener a otros, entonces la clase base que nos interesa es `ContainerControl`, derivada directamente de `ScrollableControl`. Lo que ofrece `ContainerControl`, básicamente, es la gestión del foco de entrada para los controles alojados en el nuestro.

En la práctica no es habitual usar `ContainerControl` ni `ScrollableControl` como bases directas de nuevos controles, puesto que `UserControl`, derivada de `ContainerControl`, aporta toda la funcionalidad de las demás clases y, además, Visual Basic .NET cuenta con un diseñador que facilita la creación de controles que tienen como base a `UserControl`, no ocurriendo lo mismo con las demás clases.

Nota

A los controles que usan `UserControl` como base y contienen otros se les conoce habitualmente como *controles compuestos*, mientras que los derivados de `Control` son considerados *controles simples*.

Además de diseñar nuevos controles partiendo de cualquiera de las clases comentadas hasta ahora, lógicamente también podemos crear controles basados en otros ya existentes, modificando su comportamiento o bien añadiendo alguna nueva característica. Podemos, por ejemplo, derivar una nueva clase tomando como base a `Windows.Forms.Button` creando una nueva clase de botón que, al ser pulsado, genera un pitido.

La clase Control

Dado que Control es la clase de la cual derivan todas las demás clases de controles, ya sea directa, como ScrollableControl, o indirectamente, como UserControl, está claro que su conocimiento resulta fundamental para poder crear nuestros nuevos controles.

Todos los miembros de Control estarán presentes en nuestra clase de control, indistintamente de dónde derivemos.

Ya en el capítulo dedicado a los formularios Windows conocimos muchos de los miembros de Control, miembros que entonces apuntábamos como comunes entre todos los controles que utilizamos en los diversos ejemplos. Esos miembros también aparecerán en nuestro control, sin necesidad de que hagamos nada especial. Sí nos interesa, sin embargo, saber cómo modificar ese comportamiento por defecto, por ejemplo redefiniendo métodos, ocultando propiedades, etc.

Si al utilizar un control cualquiera los miembros de Control que nos interesaban eran los públicos, accesibles desde la ventana Propiedades y el código de nuestro programa, al derivar una nueva clase tomando Control como base también serán de nuestro interés, y de manera especial, todos los miembros protegidos.

Como ya sabe, los miembros protegidos de una clase no son accesibles desde el exterior, por ejemplo al crear un objeto de dicha clase, pero sí pueden ser usados por clases derivadas.

Derivar de Control

Tras crear una nueva biblioteca con Visual Basic .NET, que nos servirá para alojar nuestros controles, para derivar una nueva clase desde Control procederíamos de la misma manera que lo hicimos en el capítulo previo con Control. Basta con tomar como base a Control para tener ya un control con cierta funcionalidad, capaz de mostrarse en el interior del formulario en una posición y con unas dimensiones, aunque sin nada en su interior.

Introduzca el código siguiente en el editor:

```
Public Class Flecha
  Inherits System.Windows.Forms.Control
End Class
```

Compile el proyecto y añada a la solución una aplicación basada en formularios Windows.

Personalice el Cuadro de herramientas añadiendo el nuevo control, como se ha hecho en la figura 16.1, y a continuación insértelo en el formulario. Observe la ventana Propiedades. En ella aparecen propiedades como BackColor y ForeColor, Font, TabStop y TabIndex, Text, etc. Todos esos miembros han sido heredados de la clase Control.

Figura 16.1. Añadimos el nuevo control al Cuadro de herramientas

Dibujo del control

Un control que dispone de todas las propiedades habituales pero que no las usa para nada, como en este momento ocurre con nuestra clase `Flecha`, no es demasiado interesante. Lo primero que haremos será dar un contenido visual a ese control, cuya funcionalidad, hasta ahora, no hemos definido. Por su nombre, no obstante, es fácil deducir que se tratará de un control gráfico que mostrará una flecha o, mejor dicho, aparecerá como una flecha en el contenedor donde sea insertado.

La clase `Control` se encarga de procesar la mayoría de los mensajes del sistema y dispositivos de entrada de información, convirtiendo esos eventos en invocaciones a métodos que nosotros, en nuestra clase derivada de `Control`, podemos redefinir. Cada vez que es necesario redibujar un control se genera el evento `Paint`, evento que la clase `Control` traduce en una invocación al método `OnPaint`. Análogamente encontramos el método `OnEnter`, `OnKeyDown`, `OnGotFocus`, etc.

Todos estos métodos pueden ser redefinidos en nuestra nueva clase de control. Para ello recurra a las listas que aparecen en la parte superior del editor de código. En la de la izquierda seleccione el elemento (**Reemplazos**), consiguiendo en la derecha la lista de todos los miembros que pueden redefinirse. Seleccionamos `OnPaint` y completamos este método con el código siguiente:

```
' Dibujo la flecha que aparecerá en el control
Protected Overrides Sub OnPaint( _
    ByVal e As System.Windows.Forms.PaintEventArgs)
  ' Ejecutamos el OnPaint de la clase base,
  ' por si es en este caso
  MyBase.OnPaint(e)
  ' Obtenemos una referencia a la propiedad
  ' Propiedad Interior
```

```
    Dim Lapiz As New Pen(Me.ForeColor, 3)

    Lapiz.EndCap = Drawing.Drawing2D.LineCap.ArrowAnchor
    Lapiz.StartCap = Drawing.Drawing2D.LineCap.ArrowAnchor

    e.Graphics.DrawLine(Lapiz, 0, _
      Me.ClientSize.Height \ 2, _
      Me.ClientSize.Width - 1, _
      Me.ClientSize.Height \ 2)
End Sub
```

Lo primero que hacemos es invocar al método `OnPaint` de la clase base. Esto es habitual al redefinir un método en una clase derivada, ya que así permitimos, en este caso, que la clase `Control` efectúe su parte del trabajo, por ejemplo obteniendo un manejador de ventana para acceder al área de dibujo. A continuación usamos los servicios de GDI+, que ya conocemos, para dibujar una línea con dos puntas de flecha en los extremos.

Simplemente volviendo a compilar la biblioteca que tiene el control, asumiendo que tiene abierta en la misma solución un proyecto Windows con un formulario que contiene uno o más controles `Flecha`, verá que automáticamente aparece el dibujo.

Modifique en la ventana Propiedades el valor de `ForeColor`, verá que de inmediato la flecha se dibuja con el nuevo color. Si altera las dimensiones iniciales del control `Flecha`, no obstante, comprobará que ocurre algo extraño. No se dibuja adecuadamente y el efecto, como se aprecia en la parte derecha de la figura 16.2, no es precisamente el deseado.

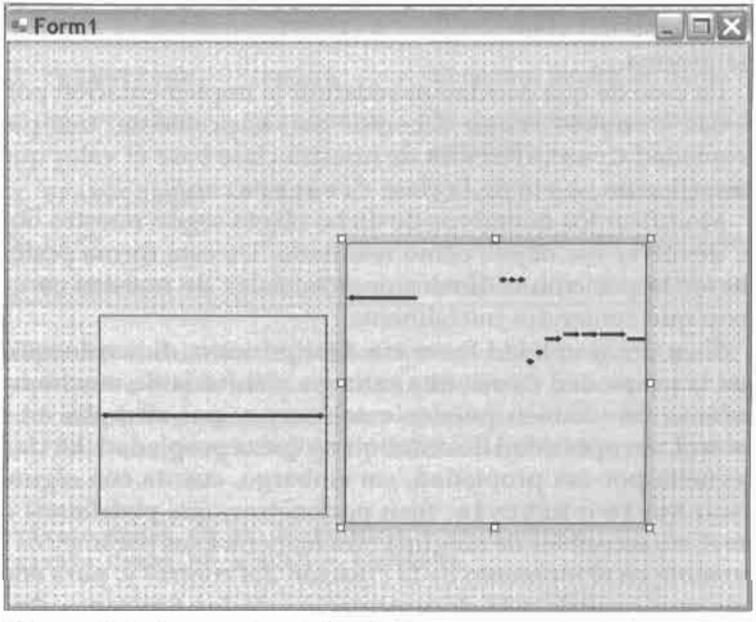


Figura 16.2. La actualización visual de nuestro control no es perfecta

Propiedades protegidas

La clase `Control` dispone de varias propiedades protegidas, así como una importante lista de métodos protegidos. Podemos tanto redefinir estos miembros como utilizarlos cuando nos interese. Una de esas propiedades es `ResizeRedraw`, que controla la actualización visual del control cuando se alteran sus dimensiones, y otra `CreateParams`, usada para recuperar diversos parámetros antes de la creación del control.

Usando la propiedad `ResizeRedraw` podemos evitar el problema que tenemos ahora mismo con nuestro control `Flecha`, codificando un constructor por defecto como el siguiente:

```
Public Sub New()
  MyBase.New()
  Me.ResizeRedraw = True
End Sub
```

Tras llamar al constructor de la clase base, `Control` en este caso, lo único que hacemos es asignar el valor `True` a la propiedad `ResizeRedraw`. Con este cambio, tras compilar de nuevo la biblioteca, podrá ver que el control actualiza su apariencia sin problemas.

La segunda propiedad mencionada, `CreateParams`, es sólo de lectura. Puede interesarnos, por tanto, a escala informativa. Su utilidad principal, sin embargo, la encontraremos al redefinirla en nuestra clase de control, puesto que podemos modificar los parámetros que `Control` entrega al sistema para la creación del control y, por lo tanto, añadir características específicas de la plataforma.

En caso de que decidamos redefinir la implementación por defecto de la propiedad `CreateParams` debemos dar, básicamente, tres pasos: recoger de la propiedad `CreateParams` de nuestra clase base el valor que contiene, una referencia a un objeto de la clase `CreateParams`.

Modificar los miembros de dicho objeto según nuestro objetivo y, finalmente, devolver ese objeto como resultado. De esta forma podemos, por ejemplo, alterar la posición o dimensiones iniciales de nuestro control o establecer el texto que contendrá inicialmente.

Si su única utilidad fuese esa, fijar posición, dimensiones o texto, lo cierto es que la propiedad `CreateParams` no resultaría de mucho interés, ya que esos parámetros también pueden establecerse, por ejemplo, en el constructor del control, sin necesidad de redefinir ninguna propiedad. La clase `CreateParams` devuelta por esa propiedad, sin embargo, cuenta con algunos miembros que, como `Style` o `ExStyle`, fijan parámetros, que podríamos considerar de bajo nivel, no accesibles de ninguna otra manera. Esos parámetros se utilizan exclusivamente en el momento de la creación del control y, para entenderlos, tendríamos que recurrir a la documentación de las funciones `CreateWindow()` y `CreateWindowEx()` del API de Windows.

El miembro `Style` de `CreateParams` es un entero que puede contener múltiples bits activos indicando diversos estilos para el control. Dichos estilos se corresponden con el parámetro `style` de la función `CreateWindow()`. Algunos de esos parámetros son los que se enumeran en la tabla 16.1. Lo mismo ocurre con el miembro `ExStyle`, aunque en este caso se corresponde con un parámetro de la función `CreateWindowEx()` que se usa para algunos estilos extendidos. En la tabla 16.2 se citan algunos de los estilos extendidos.

Tabla 16.1. Valores que puede contener el miembro `Style` de `CreateParams`

Constante Win32	Valor	Significado
WS_BORDER	&H800000	Dota al control de un borde simple alrededor
WS_HSCROLL	&H100000	Añade una barra de desplazamiento horizontal al control
WS_VSCROLL	&H200000	Añade una barra de desplazamiento vertical al control
WS_TICKFRAME	&H40000	Dota al control de un borde grueso redimensionable
WS_MINIMIZEBOX	&H10000	Muestra en la ventana un botón para minimizar
WS_MAXIMIZEBOX	&H20000	Muestra en la ventana un botón para maximizar

Tabla 16.2. Valores que puede contener el miembro `ExStyle` de `CreateParams`

Constante Win32	Valor	Significado
WS_EX_CLIENTEDGE	&H200	El control tendrá un borde resaltado (3D)
WS_EX_STATICEDGE	&H200000	El control tendrá un borde tridimensional estático
WS_EX_TRANSPARENT	&H20	El control será transparente
WS_EX_LEFT	0	La alineación del control será a izquierda
WS_EX_RIGHT	&H1000	La alineación del control será a derecha

Aunque las constantes indicadas son sólo una pequeña parte de las que tenemos a nuestra disposición, resultan suficientes para mejorar la apariencia de nuestro control. Podemos, por ejemplo, añadirle un borde alrededor o, incluso mejor, definir una propiedad que permita elegir el tipo de borde:

```
Private Const WS_EX_STATICEDGE = &H20000
Private Const WS_BORDER = &H800000

Private FBorderStyle As BorderStyle = BorderStyle.None
```

```
<Category("Apariencia"), _
 Description("Fija el tipo de borde alrededor del control"), _
 DefaultValue(Windows.Forms.BorderStyle.None)> _
Public Property BorderStyle() As Windows.Forms.BorderStyle
  Get
    Return FBorderStyle
  End Get
  Set(ByVal Value As Windows.Forms.BorderStyle)
    FBorderStyle = Value


    Me.RecreateHandle()
  End Set
End Property
```

Observe que han sido declaradas dos constantes con los valores apuntados antes en las tablas 16.1 y 16.2, puesto que dichas constantes no se encuentran definidas en Visual Basic .NET. A continuación hemos declarado la variable que contendrá el valor de la propiedad y definido ésta. Lo único destacable de `BorderStyle` es el hecho de que al modificar el valor se invoca al método `RecreateHandle()`.

No basta con usar `Invalidate()` para forzar la actualización visual del control. `Invalidate()` lo único que hace es forzar la generación de un evento `Paint`, pero la modificación de los estilos de un control sólo se produce, según se ha dicho ya antes, en el momento de la creación. Llamando a `RecreateHandle()` conseguimos la destrucción y recreación del control, lo cual implica la recuperación de los estilos de la propiedad `CreateParams`. Ésta deberemos redefinirla dejándola como sigue:

```
Protected Overrides ReadOnly Property _
    CreateParams() As CreateParams
  Get
    Dim Parametros As CreateParams = MyBase.CreateParams()

    Select Case FBorderStyle
      Case BorderStyle.FixedSingle

        Parametros.Style = Parametros.Style Or WS_BORDER
      Case BorderStyle.Fixed3D
        Parametros.ExStyle = _
           Parametros.ExStyle Or WS_EX_STATICEDGE
    End Select

    Return Parametros
  End Get
End Property
```

Fíjese en que no asignamos directamente el estilo deseado al miembro `Style` o `ExStyle`, según proceda, de `CreateParams`, sino que lo añadimos mediante el operador lógico `Or`. De esta forma preservamos los valores que pudiese

haber asignado la clase `Control` en su propia implementación de la propiedad. El resultado, tras estos cambios, será una versión del control en la cual podemos personalizar el tipo de borde que aparece alrededor. En la figura 16.3 puede ver tres controles `Flecha` en el interior de un formulario, uno sin borde, otro con un borde simple y el tercero con un borde tridimensional habitual en los controles estáticos que no pueden tomar el foco de entrada.

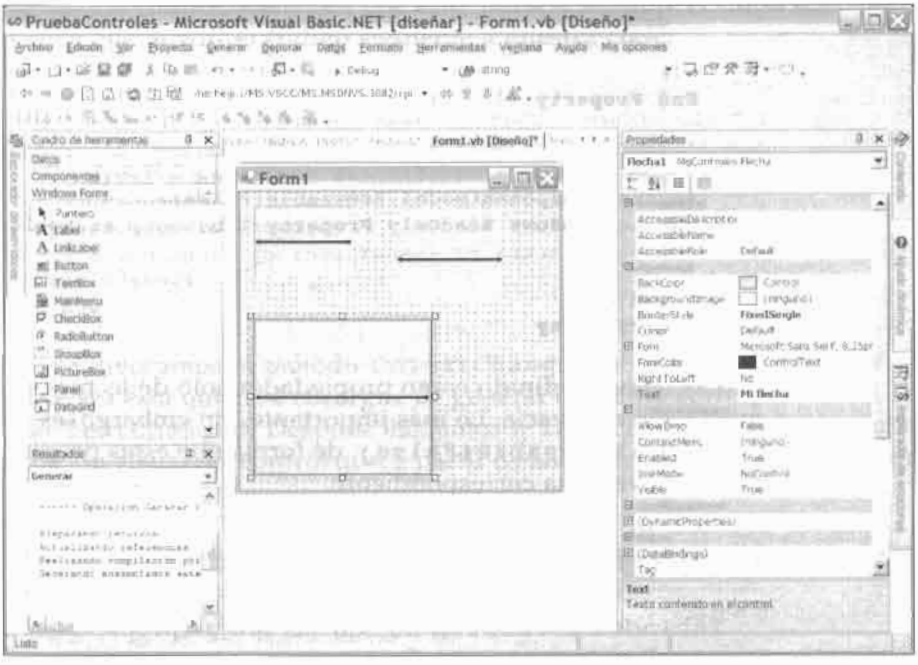


Figura 16.3. El mismo control con diferentes tipos de borde

Ocultación de propiedades

Nuestro control no está diseñado para tomar el foco de entrada, de hecho no puede tomarlo y por ello hemos usado como base la clase `Control` y no otra de las citadas previamente. Incluso podemos indicarlo visualmente usando el borde tridimensional que acabamos de añadir en el punto anterior. A pesar de todo, en la ventana **Propiedades** siguen apareciendo propiedades como `TabStop` y `TabIndex`, relacionadas precisamente con el acceso al control mediante el tabulador.

Ya sabemos que podemos controlar la aparición de una propiedad en la ventana **Propiedades** mediante el atributo `System.ComponentModel.Browsable`. El problema es que la propiedad ya existe en la clase base y, además, en este caso concreto no son propiedades `Overridable`, es decir, que podamos redefinir.

Siempre tenemos la opción, sin embargo, de ocultar un miembro de una clase base definiendo otro idéntico y utilizando la palabra clave `Shadows`.

Dado que lo que no nos interesa, para el control `Flecha`, es que aparezcan las propiedades `TabStop` y `TabIndex`, podríamos ocultarlas añadiendo el código siguiente a nuestra propia clase.

```
<System.ComponentModel.Browsable(False)> _
Public Shadows ReadOnly Property TabIndex() As Integer
  Get

  End Get
End Property

<System.ComponentModel.Browsable(False)> _
Public Shadows ReadOnly Property TabStop() As Boolean
  Get

  End Get
End Property
```

Las hemos definido como propiedades sólo de lectura y, además, la implementación está vacía. Lo más importante, sin embargo, es que hemos aplicado el atributo `Browsable(False)`, de forma que estas propiedades ya no aparecen en la ventana correspondiente.

Eventos de cambios en propiedades

`TabIndex` y **`TabStop`** no son las únicas propiedades no útiles con que cuenta el control `Flecha`. En este momento tampoco son útiles propiedades como **`Text`** y **`Font`**, por poner un ejemplo, ya que el control tan sólo muestra una línea en su interior. Podríamos ocultarlas también, pero en lugar de hacerlo vamos a aprovecharlas para mostrar un título encima de la flecha. Lo único que tenemos que hacer será añadir la sentencia siguiente al final del método `OnPaint()` que habíamos redefinido previamente.

```
e.Graphics.DrawString(Me.Text, Me.Font, Brushes.Black, 0, 0)
```

Tras compilar de nuevo la biblioteca que contiene el componente compruebe su comportamiento. Si modifica la propiedad `Font` verá que, de inmediato, el texto que muestra el control se actualiza convenientemente. No ocurre lo mismo, sin embargo, cuando se modifica la propiedad `Text`. Está claro que la implementación que se hace en `Control` de la propiedad `Font` cuenta con una llamada a `Invalidate()`, mientras que la asignación a la propiedad `Text` simplemente conserva el nuevo valor.

La mayoría de las propiedades de la clase `Control` generan un evento cada vez que se modifica su valor, facilitando así la actualización de los elementos dependientes que pudieran existir. *En realidad desde el apartado `Set` de la* propiedad lo que se hace es invocar a un método cuyo nombre sigue el patrón

`On*Propiedad*Changed()`, sustituyendo *`Propiedad`* por el nombre de la propiedad que corresponda. Dichos métodos son protegidos, de tal forma que el usuario de un control tendría que utilizar forzosamente el evento `TextChanged`, `EnabledChanged`, `FontChanged` o, en definitiva, aquél que corresponda a la propiedad que desea controlar. En nuestra clase de control, sin embargo, tenemos acceso a los miembros protegidos y, por tanto, podemos redefinirlos.

Para conseguir que una modificación de la propiedad `Text` actualice la visualización del control, haciendo que éste muestre siempre el título adecuado, tendríamos que añadir el código siguiente a nuestra clase:

```
Protected Overrides Sub OnTextChanged( _
      ByVal e As System.EventArgs)
  MyBase.OnTextChanged(e)
  Invalidate()
End Sub
```

Primero invocamos al método `OnTextChanged()` de la clase base, de tal forma que sea ésta quien se encargue de generar el evento correspondiente, en este caso `TextChanged`. Después llamamos a `Invalidate()` para forzar la actualización visual del control que, a partir de ahora, tendrá un funcionamiento más adecuado.

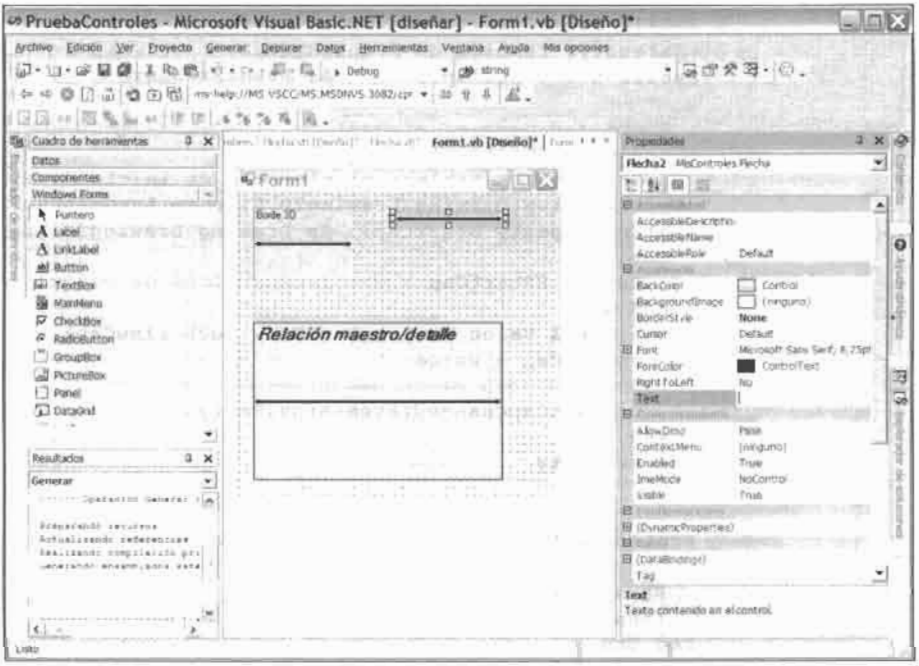


Figura 16.4. Varios controles `Flecha` con y sin título

Nuevas propiedades y eventos

Además de modificar el comportamiento por defecto de la clase `Control` redefiniendo y ocultando miembros, que es, básicamente, lo que hemos hecho hasta ahora, lógicamente también podemos ampliarlo añadiendo miembros nuevos. Ya hemos definido en la clase `Flecha` una nueva propiedad, llamada `BorderStyle` y, de igual forma, podemos implementar nuevos métodos y eventos. El procedimiento lo conocemos sobradamente.

En su diseño actual nuestro control siempre dibuja una flecha a ambos extremos de una línea recta. Podríamos habilitar las propiedades necesarias para que este aspecto fuese configurable y el usuario del control pudiese elegir lo que desea ver en los extremos, de igual forma que elige el color o el tipo de letra del título.

Tendríamos que definir dos propiedades, `StartCap` y `EndCap`, y de paso facilitar también los eventos correspondientes a cambios en su valor. El código que tendríamos que añadir a nuestra clase sería el mostrado a continuación.

```
Private FStartCap As Drawing.Drawing2D.LineCap = _
   Drawing.Drawing2D.LineCap.ArrowAnchor
Private FEndCap As Drawing.Drawing2D.LineCap = _
   Drawing.Drawing2D.LineCap.ArrowAnchor


<Category("Cambios de propiedad")> _
Public Event StartCapChanged As EventHandler
<Category("Cambios de propiedad")> _
Public Event EndCapChanged As EventHandler


<Category("Apariencia"), _
 Description("Establece el elemento de inicio de línea"), _
 DefaultValue(Drawing.Drawing2D.LineCap.ArrowAnchor)> _
Public Property StartCap() As Drawing.Drawing2D.LineCap
  Get
    Return FStartCap
  End Get
  Set(ByVal Value As Drawing.Drawing2D.LineCap)
    FStartCap = Value

    OnStartCapChanged(EventArgs.Empty)
  End Set
End Property


Protected Sub OnStartCapChanged(ByVal e As System.EventArgs)

  RaiseEvent StartCapChanged(Me, e)
  Invalidate()
End Sub
```

```
<Category("Apariencia"), _
 Description("Establece el elemento de fin de línea"), _
 DefaultValue(Drawing.Drawing2D.LineCap.ArrowAnchor)> _
Public Property EndCap() As Drawing.Drawing2D.LineCap
  Get
    Return FEndCap
  End Get
  Set(ByVal Value As Drawing.Drawing2D.LineCap)
    FEndCap = Value
    OnEndCapChanged(EventArgs.Empty)
  End Set
End Property

Protected Sub OnEndCapChanged(ByVal e As System.EventArgs)
  RaiseEvent EndCapChanged(Me, e)
  Invalidate()
End Sub
```

Hemos seguido el mismo esquema usado en la propia clase `Control`, invocando desde el apartado `Set` de las propiedades a los métodos `OnStartCapChanged()` y `OnEndCapChanged()`. Estos métodos son protegidos, accesibles para cualquier clase que pudiera derivarse de `Flecha`. En su interior se genera el evento de cambio de valor y se invalida el control para forzar la actualización.

De manera análoga, puede mejorar el control implementando una propiedad que nos permita definir el grosor de la línea o el tipo de trazo, así como su orientación que, ahora mismo, siempre es horizontal.

Figura 16.5. Ahora podemos elegir el elemento que aparecerá en los extremos de la línea

Interceptar los eventos de teclado y ratón

La clase `Control` se encarga de procesar, como se indicó al principio, todos los mensajes que recibe del sistema y de dispositivos como el teclado y el ratón. En el caso de estos dispositivos, lo habitual es que genere el evento correspondiente, de forma que el usuario del control pueda responder adecuadamente a las pulsaciones de teclado o botones del ratón. A veces, no obstante, podemos necesitar incluir en nuestro control código asociado a dichos eventos.

El mecanismo es idéntico al descrito antes para los eventos de notificación de cambios en los valores de las propiedades. Existe un método protegido, que podemos sobrecargar, encargado de generar los eventos de teclado y ratón. Bastaría con añadir el código siguiente a nuestra clase para conseguir que cada vez que se pulse sobre un control `Flecha`, con el botón principal del ratón, se inviertan los valores de las propiedades `StartCap` y `EndCap`, lo cual tendrá un reflejo inmediato en la apariencia en caso de que dichas propiedades indicasen distinto tipo de elemento.

```
Protected Overrides Sub OnMouseUp(
   ByVal e As System.Windows.Forms.MouseEventArgs)
  If e.Button = MouseButtons.Left Then
    Dim Extremo As Drawing.Drawing2D.LineCap = StartCap
    StartCap = EndCap
    EndCap = Extremo
  End If
  MyBase.OnMouseUp(e)
End Sub
```

De manera análoga podríamos interceptar cualquier otro evento de teclado o ratón, tan sólo hay que redefinir el método apropiado y, como sabe, el editor de código nos ofrece en la lista desplegable que aparece en la parte superior derecha una lista de todos esos métodos.

Controles compuestos

El control que hemos desarrollado como ejemplo en los puntos anteriores, `Flecha`, hereda gran parte de su funcionalidad de la clase `Control`, lo hemos visto, pero no utiliza otros componentes para efectuar su trabajo. Hay ocasiones en las que más que desarrollar un control totalmente nuevo, como es el caso de `Flecha`, nos interesará componer un control complejo a partir de otros ya existentes. En estos casos la base adecuada es `UserControl` que, como vamos a experimentar de inmediato, cuenta con diseñadores que facilitan su desarrollo hasta situarlo al mismo nivel que una interfaz de usuario basada en un formulario.

La clase `UserControl`, como ya sabe, está derivada de `ContainerControl` que, a su vez, deriva de `ScrollableControl` que, a su vez, tiene como base a `Control`. Esto significa que todo lo que hemos aprendido hasta ahora nos sigue siendo útil, es más, podríamos decir que indispensable, a la hora de crear controles de usuario o derivados de `UserControl`. Esta clase añade los elementos necesarios para hacer posible la navegación por los componentes *hijo* que contiene, gestionando el foco de entrada adecuadamente.

Veamos de manera rápida cómo crear un control de este tipo basándonos en un sencillo ejemplo. Comenzaremos añadiendo a la misma biblioteca donde tenemos a `Flecha` un control de usuario, utilizando para ello el cuadro de diálogo que aparece en la figura 16.6. En la lista de la derecha hemos elegido el elemento Control de usuario, mientras que en la parte inferior hemos introducido el nombre del nuevo control: `EuroCalc`.

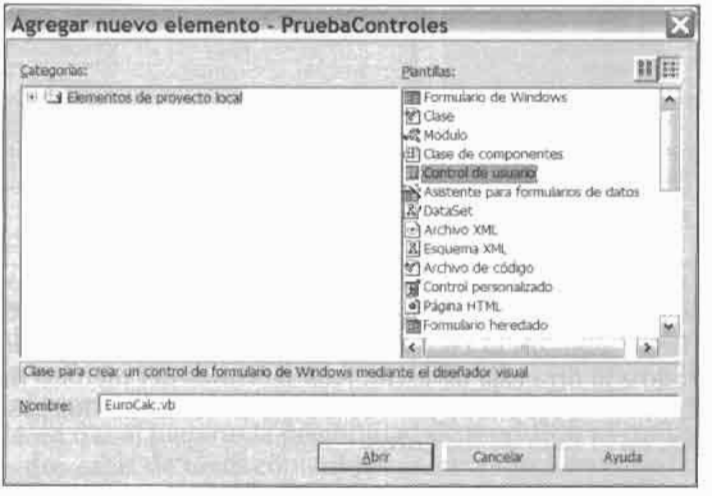


Figura 16.6. Añadimos un control de usuario a la biblioteca actual de controles Windows

De inmediato aparecerá un diseñador similar al utilizado para los formularios Windows, aunque en este caso no existe un borde ni tampoco una barra donde aparezca el título y los botones de cierre, maximizado y minimizado. En dicho contenedor podemos insertar los controles que deseemos, utilizando el Cuadro de herramientas, personalizándolos mediante la edición de propiedades y asociación de código a sus eventos.

En resumen, podríamos decir que el trabajo es idéntico al del desarrollo de una interfaz de usuario, pero el resultado no es una aplicación sino un control reutilizable.

Observe la figura 16.7. Corresponde al control que hemos creado antes tras añadir dos cajas de texto y dos etiquetas, que hemos personalizado modificando propiedades como `Text`, `TextAlign` y `Anchor`. La apariencia final es la de una calculadora con doble pantalla, como las que se han hecho tan populares con la disponibilidad del euro.

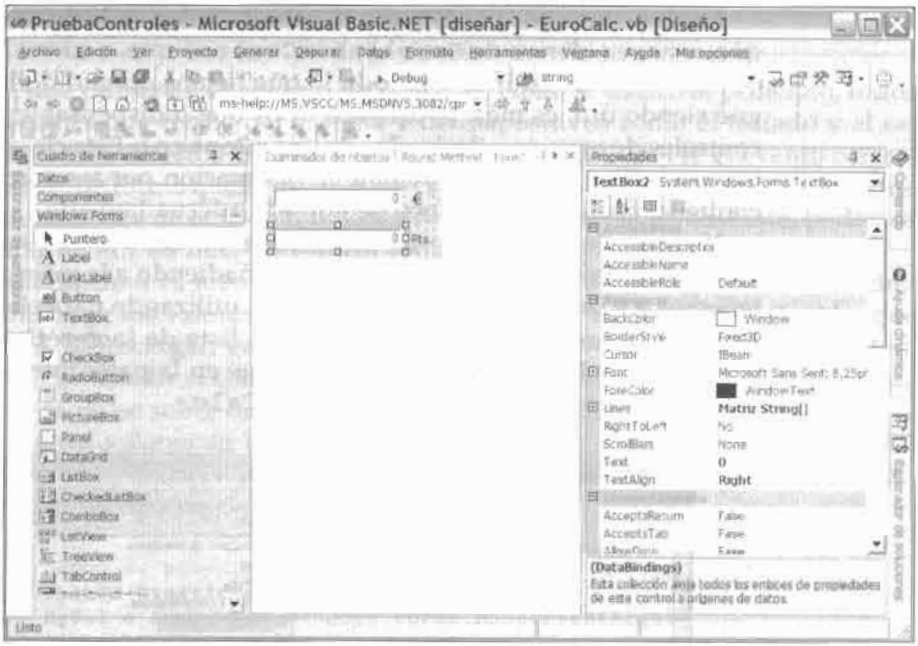


Figura 16.7. Diseñamos el control de una forma visual

El único código de este control estará asociado al evento `KeyPress` de las dos cajas de texto. Como puede ver a continuación, comprobamos si se ha pulsado **Intro**, caso en el cual hacemos el cálculo de conversión y asignamos el resultado a la otra caja de texto. Nos servimos del método `Math.Round()` para redondear el resultado sin decimales, en el caso de las pesetas, o con dos decimales, en el caso de los euros.

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.KeyPressEventArgs) _
  Handles TextBox1.KeyPress
  If e.KeyChar = vbCr Then

    TextBox2.Text = Math.Round(CDbl(TextBox1.Text) * 166.386)
  End If
End Sub

Private Sub TextBox2_KeyPress(ByVal sender As Object, _
ByVal e As System.Windows.Forms.KeyPressEventArgs) _
Handles TextBox2.KeyPress
  If e.KeyChar = vbCr Then

    TextBox1.Text = Math.Round(CDbl(TextBox2.Text) / 166.386, 2)
  End If
End Sub
```

No tenemos que hacer nada más. Basta con compilar de nuevo la biblioteca en la que se encuentran nuestros controles y, después, insertar un `EuroCalc`

en un formulario. Para comprobar el funcionamiento tendrá que ejecutar el proyecto, ya que este nuevo control no incorpora propiedades que permitan asignar o recuperar los valores en la fase de diseño. En la figura 16.8 aparecen dos controles `EuroCalc` en funcionamiento.

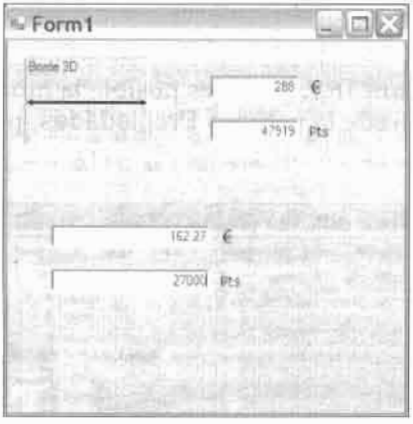


Figura 16.8. Nuestro nuevo control en funcionamiento

Propiedades de componentes contenidos

Como puede ver en el control `EuroCalc`, las propiedades y, en general, los miembros de los controles contenidos en él no son directamente accesibles desde fuera. Para hacerlos visibles tendríamos que codificar miembros que facilitasen el acceso a los miembros de los controles contenidos. Por ejemplo, suponga que desea dar al usuario la posibilidad de modificar el contenido de cualquiera de las dos cajas de texto contenidas en el control `EuroCalc`, de tal forma que pueda efectuar una asignación en la fase de diseño o bien mediante código. Tan sólo tendríamos que añadir las dos propiedades siguientes:

```
<RefreshProperties(RefreshProperties.Repaint)> _
Public Property Euros() As Double
  Get
    Return TextBox1.Text
  End Get
  Set(ByVal Value As Double)

    TextBox1.Text = Value

    TextBox2.Text = Math.Round(CDbl(TextBox1.Text) * 166.386)
  End Set
End Property
```

```
<RefreshProperties(RefreshProperties.Repaint)>
Public Property Pesetas() As Double
```

```
    Get
      Return TextBox2.Text
    End Get
    Set(ByVal Value As Double)
      TextBox2.Text = Value
      TextBox1.Text = Math.Round(CDbl(TextBox2.Text) / 166.386, 2)
    End Set
  End Property
```

Como se aprecia en la figura 16.9, ahora es posible la modificación del contenido de las cajas de texto desde la ventana **Propiedades**, posibilidad que antes no teníamos.

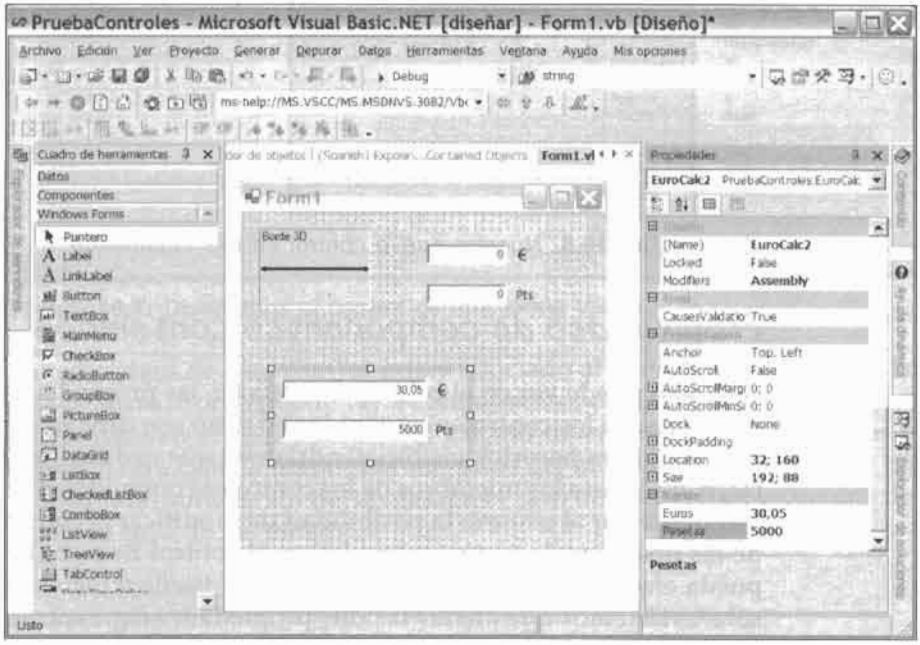


Figura 16.9. Editamos las nuevas propiedades del control `EuroCalc`

En lugar de definir una propiedad en `EuroCalc` para cada propiedad de los controles que lo constituyen, lo cual nos ofrece un cierto control sobre los elementos a los que tendrá acceso el usuario del componente, podemos definir una propiedad que ofrezca acceso directo a todo el control. Podríamos implementar dicha propiedad así:

```
Public ReadOnly Property EurosControl() As TextBox
  Get
    Return TextBox1
  End Get
End Property
```

La propiedad es sólo de lectura, de forma que no pueda asignar a `EurosControl` una caja de texto diferente a las que ya hay insertadas en `EuroCalc`. Al

devolver una referencia a esa caja de texto, sin embargo, estamos permitiendo el acceso a todos sus miembros. En la ventana **Propiedades** la propiedad `EurosControl` aparece con una marca que indica la existencia de subpropiedades, en realidad la lista de propiedades del `TextBox`. De esta forma podemos modificar la alineación, tipo de letra, etc. En la figura 16.10 puede ver la lista de propiedades y subpropiedades.

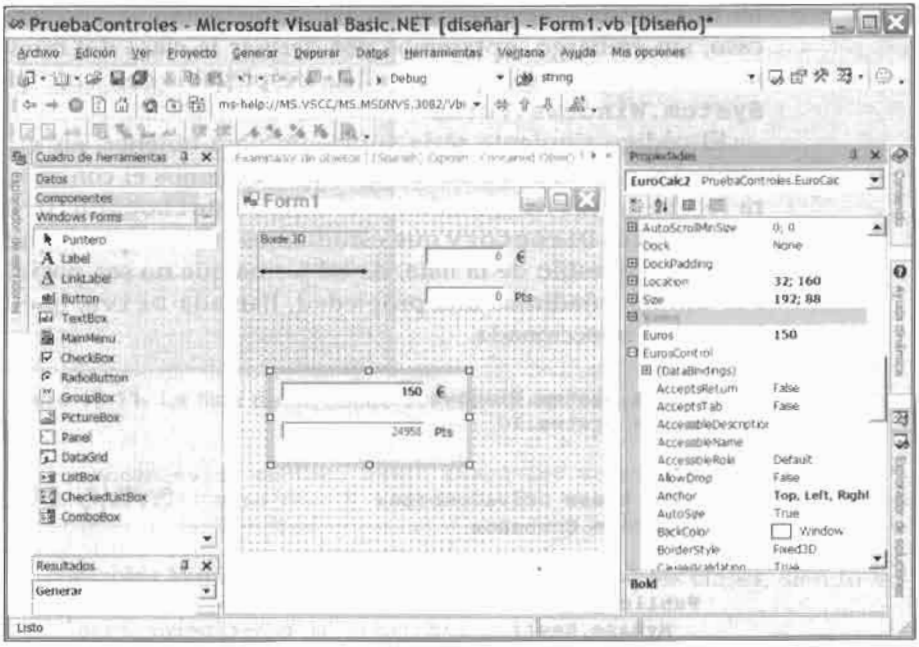


Figura 16.10. Las propiedades de la caja de texto aparecen en la ventana Propiedades

Nota

Recuerde que puede servirse de los eventos `PropiedadChanged` para detectar el cambio de cualquier propiedad de la caja de texto y actuar en consecuencia.

Otras bases para nuestros controles

Las clases `Control` y `UserControl` no son las únicas que pueden usarse como base para crear nuevos controles. En realidad, podemos crear un control partiendo de cualquier clase existente en `System.Windows.Forms`, cualquier control que ya exista, independientemente de su simplicidad o complejidad. Usando las técnicas descritas en éste y el capítulo anterior podemos modificar

el comportamiento de cualquier control, añadiendo nuevas propiedades y eventos, ocultando miembros, alterando su estilo, etc.

Suponga que quiere crear un control que aparezca como una lista desplegable de la que el usuario pueda seleccionar una de las unidades de disco disponibles en el sistema. Sería absurdo crear toda la funcionalidad de una nueva lista desplegable partiendo de la clase `Control`. Tampoco necesitamos toda la funcionalidad de `UserControl`, aunque sería muy fácil insertar en el diseñador un `ComboBox` y aprovechar un evento para añadir los elementos. En nuestro caso, sin embargo, optaremos por derivar la clase del nuevo control directamente de `ComboBox`, que es la lista desplegable corriente que encontramos en `System.Windows.Forms`.

El código siguiente sería suficiente para obtener, en principio, la lista de unidades y facilitar su selección. Aprovechamos el constructor de la clase para recuperar esa lista, mediante el método `GetLogicalDrives()` de la clase `System.IO.Directory` que estudiamos en un capítulo previo. También modificamos el estilo de la lista, de tal forma que no sea posible teclear un texto. Por último, añadimos una propiedad, llamada `Drive`, que facilita el acceso a la unidad seleccionada.

```
Imports System.Windows.Forms
Imports System.IO

Public Class DriveComboBox
  Inherits ComboBox

  Public Sub New()
    MyBase.New()

    Me.DropDownStyle = ComboBoxStyle.DropDownList
    Dim Unidad As String
    Me.Items.Clear()

    For Each Unidad In Directory.GetLogicalDrives
      Me.Items.Add(Unidad)
    Next

    Me.Text = Me.Items(0)
  End Sub



  Public Property Drive() As String
    Get
      Return Me.Text
    End Get
    Set(ByVal Value As String)
      Me.Text = Value
    End Set
  End Property
End Class
```

En la figura 16.11 puede observar el aspecto del nuevo control. En la práctica sería necesario añadir propiedades y eventos que controlasen la modificación de la propiedad `Drive` y, posiblemente, habría que ocultar propiedades como `Text` y `DropDownStyle`. Para todo ello ya cuenta con los conocimientos necesarios.

Figura 16.11. La lista de unidades del control `DriveComboBox`

Puntos clave

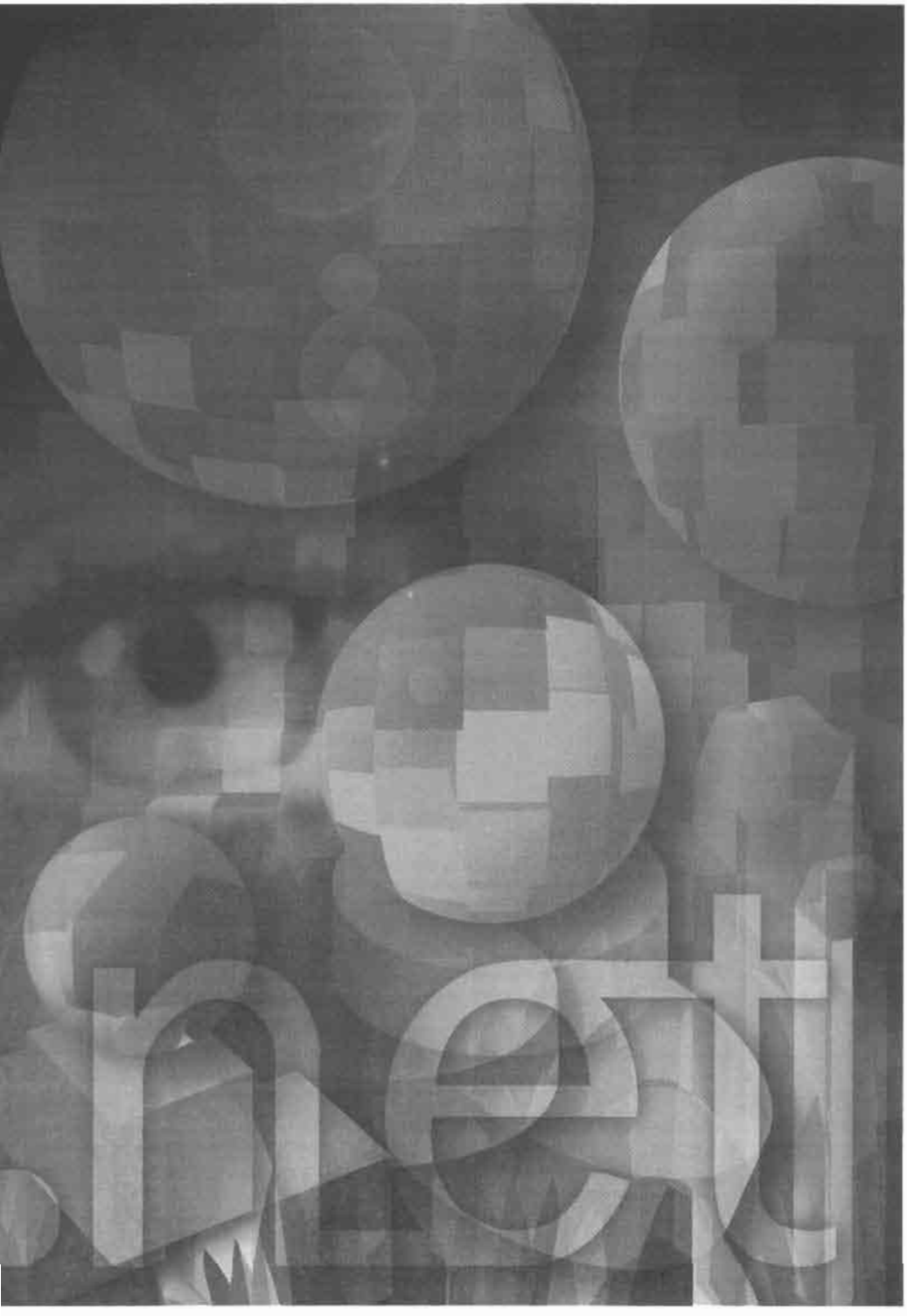
- Los controles Windows pueden derivarse de múltiples clases, siendo las principales `Control` y `UserControl`.
- Cuando el control a desarrollar es simple, no compuesto de otros, y se encarga de dibujarse a sí mismo en el contenedor, la base apropiada es la clase `Control`.
- Para diseñar un control que está compuesto de otros usaremos como base `UserControl`, clase para la que existe un diseñador prácticamente idéntico al de los formularios Windows.
- La clase `Control` dispone de un importante número de miembros protegidos, accesibles desde cualquier nuevo control que podamos desarrollar.
- Redefiniendo la propiedad sólo de lectura `CreateParams` podemos alterar el estilo del control, utilizando para ello indicadores de bajo nivel propios del API de Windows.
- Aprovechando los métodos protegidos de `Control` es posible responder a eventos de teclado y ratón, detectar cambios en propiedades y dibujar el control cuando es necesario.
- Tomando como base una clase de control ya existente podemos alterar su comportamiento por defecto, por ejemplo extendiendo sus posibilidades mediante nuevos miembros.

Resumen

Al finalizar este capítulo hemos adquirido los conocimientos necesarios para desarrollar cualquier tipo de control para formularios Windows, controles que tienen una interfaz de usuario y aparecen visualmente en el contenedor. También hemos aprendido a aprovechar muchas de las características de `Control`, clase que actúa como base de todos los controles del ámbito `System.Windows.Forms`.

El diseño de controles compuestos es muy similar al existente en versiones previas de Visual Basic. Incluso la clase que actúa como base, `UserControl`, se denominaba igual entonces. La composición de estos controles resulta muy sencilla ya que, en definitiva, se trata de insertar otros controles y utilizarlos como haríamos en cualquier aplicación.

Para terminar este trío de capítulos, dedicados a la creación de componentes y controles, en el próximo abordaremos la creación de controles para formularios Web, controles que generan código HTML o están compuestos de código HTML.



17
Controles Web

De igual manera que podemos crear controles específicos para su uso en formularios Windows, también existe la posibilidad de crear controles para formularios Web, controles que, por la propia naturaleza de la aplicación a la que van destinados, resultan muy distintos de los que hemos conocido en el capítulo previo. No hay más que tener en cuenta que el control propiamente dicho se ejecuta en un servidor web, no en un cliente. No existe, por tanto, la posibilidad de acceso directamente a un área de dibujo.

Los controles Web podemos clasificarlos básicamente en dos categorías: controles de servidor y *pagelets*. Los primeros, como vamos a ver a continuación, se desarrollan derivando de una cierta clase y escribiendo código, como los controles para Windows. Los segundos, los *pagelet*, en realidad son fragmentos de código HTML a los cuales se les asigna un identificador que facilita su posterior reutilización. Unos y otros resultan interesantes.

La clase `Control` y sus derivadas

Todas las clases que nos interesan en este capítulo, de cara a crear nuestros propios controles web, se encuentran alojadas en el ámbito `System.Web.UI`. La que podríamos considerar raíz de todas las demás, relacionadas con los controles, es `Control`. Aunque su nombre coincide con la clase `Control` de `System.Windows.Forms`, tenga en cuenta que son clases muy diferentes y que cada una está alojada en un ensamblado distinto y un ámbito propio.

La clase `Control` dispone de los elementos necesarios para contener a otros controles, así como la capacidad de dibujarse enviando al cliente, mediante un intermediario, el código HTML apropiado. Podríamos decir que crear un nuevo control derivando de `System.Web.UI.Control` sería equivalente a tomar como base `System.Windows.Forms.Control`, tan sólo hay que redefinir un método, llamado `Render()` en el primer caso y `Paint()` en el segundo, para definir la apariencia o contenido del control en su contenedor.

Una de las clases derivadas de `Control` es `LiteralControl`. Aunque no es habitual derivar nuevas clases de ésta, dada su simplicidad, nada nos impide hacerlo. Los controles de esta clase representan elementos de contenido que no necesitan ser procesados en el servidor, es decir, son elementos estáticos, que pueden mantenerse siempre en el documento enviado al cliente.

Derivada directamente de `Control`, la clase `WebControl` es, a su vez, base de la mayoría de controles web que tenemos a nuestro alcance en el Cuadro de herramientas, como `Calendar`, `Button` o `TextBox`. Esta clase añade, a los miembros heredados de `Control`, propiedades que gestionan la apariencia del control, como `BackColor`, `BorderStyle` o `Font`, así como los métodos protegidos necesarios para personalizar el comportamiento desde clases que nosotros pudiéramos derivar.

Por último nos encontramos con la clase `UserControl`, derivada de `TemplateControl` que, a su vez, está derivada de `Control`. Al igual que ocurría con la clase `UserControl` de `System.Windows.Forms`, ésta se utiliza para crear controles compuestos de elementos ya existentes, en cierta forma creando una página o documento que podrá insertarse como unidad indisoluble en la página web que vaya a enviarse al cliente.

Desarrollo de un control simple

Veamos cómo crear un control web simple derivándolo directamente de la clase `Control`. La finalidad de este control será simplemente la de introducir una firma en el documento, de tal forma que cada vez que ésta sea necesaria no habrá más que tomar el control e insertarlo en la página. Lo único que debemos hacer es redefinir el método `Render()` heredado de `Control`, usándolo para insertar el código HTML apropiado en el único parámetro que se recibe.

Inicie una biblioteca de clases e inserte en ella el código mostrado a continuación:

```
Imports System.ComponentModel
Imports System.Web.UI


Public Class Firma
  Inherits Control


  Protected Overrides Sub Render( _
     ByVal writer As System.Web.UI.HtmlTextWriter)
```

```
    writer.Write("<pre>(c) 1997-2002 <b>Torre de Babel</b></pre>")
  End Sub
End Class
```

El parámetro que recibe el método `Render()`, de clase `HtmlTextWriter`, dispone de un método `Write()` similar al de la clase `HttpResponse` que ya usamos en el capítulo dedicado a los formularios web. Dicho método dispone de varias versiones sobrecargadas que aceptan distintos parámetros. En este caso nos limitamos a facilitar una cadena de caracteres con unas marcas HTML y un texto. Tras compilar este componente y añadirlo al Cuadro de herramientas, pruebe a insertarlo en un formulario web. Lo primero que notará es que no puede colocarlo en una posición fija del formulario, a pesar de que sí es posible hacerlo con otros controles. Esto se debe a que la clase `Control`, de la que hemos derivado nuestro control, no facilita las propiedades necesarias para mantener su posición y dimensiones.

Posición, dimensiones y estilo

Aunque podríamos añadir manualmente a nuestro control propiedades como `Height`, `Width`, `Font` y similares, lo cierto es que resulta mucho más fácil cambiar la clase base, que actualmente es `Control`, por `WebControl`. Esta clase está definida en el ámbito `System.Web.UI.WebControls`, no en `System.Web.UI`, y es la base común de todos los controles existentes en dicho ámbito, en general todos los controles que tienen un proceso en el servidor.

Sólo cambiando **`Inherits`** `Control` por **`Inherit`** `WebControls.WebControl`, y volviendo a compilar la biblioteca en la que se encuentra el control, podrá ver que el comportamiento del control cambia de inmediato. Ahora sí que podremos colocarlo en la posición que nos convenga, como cualquier otro de los controles disponibles en el Cuadro de herramientas, gracias a la funcionalidad heredada de `WebControl`. En la figura 17.1 puede ver el control insertado en un formulario web.

En este momento el control no utiliza los valores alojados en la mayoría de las propiedades heredadas de `Control`. Así, podemos modificar aspectos como el color del texto o el tamaño del tipo de letra sin que `Firma` adecúe convenientemente el código generado que, en definitiva, definirá su aspecto en el documento HTML. Lo único que tenemos que hacer es escribir el código apropiado, dejando el método `Render()` como aparece a continuación:

```
Protected Overrides Sub Render( _
   ByVal writer As System.Web.UI.HtmlTextWriter)
  writer.Write("<font color=" & Me.ForeColor.ToString & _
      " size= " & Me.Font.Size.ToString & ">")
  writer.Write( _
     "<pre>(c) 1997-2002 <b>Torre de Babel</b></pre></font>")
End Sub
```

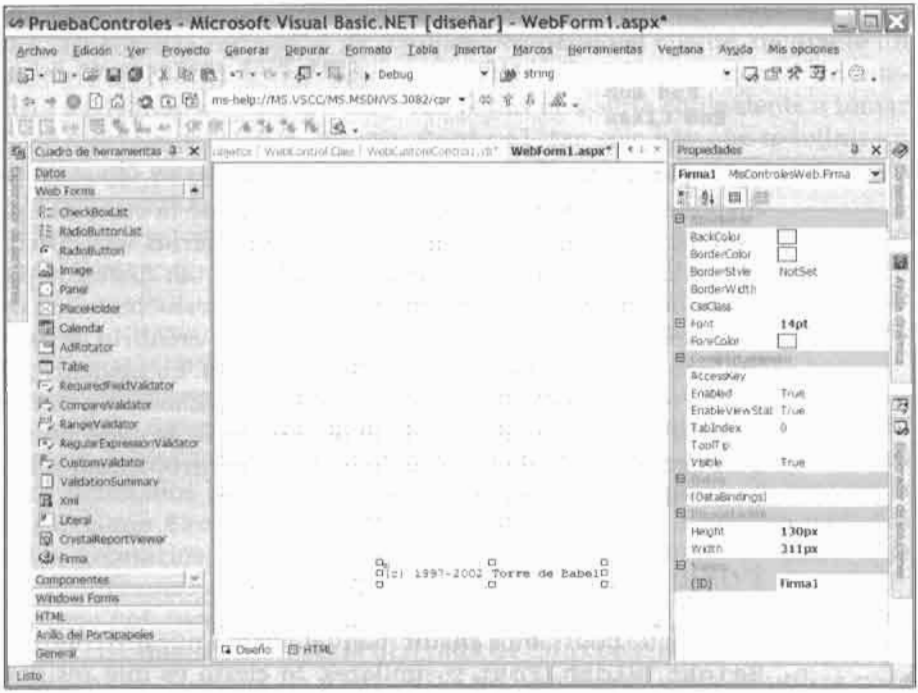


Figura 17.1. Aspecto de nuestro control insertado en un formulario web

Como puede ver, recuperamos los valores de las propiedades `ForeColor` y `Font.Size` de nuestro control para componer una etiqueta `<font>` de HTML con los atributos necesarios. Ahora es posible personalizar tanto el color como el tamaño del control.

De manera análoga puede implementarse también el uso del resto de las propiedades.

Adición de nuevos miembros

Lógicamente, podemos añadir a cualquier nueva clase de control Web los miembros que necesitemos: propiedades, métodos, eventos y miembros de datos. El sistema es exactamente el mismo que ya conocemos, incluyendo la asociación de atributos con esos miembros o con la propia clase para determinar cuál es la propiedad o el efecto por defecto. No necesitamos, por lo tanto, aprender nada nuevo.

Nota

Cuando las propiedades o métodos afecten a la apariencia del control, que es lo habitual, será entonces preciso actualizar la implementación del método `Render()`.

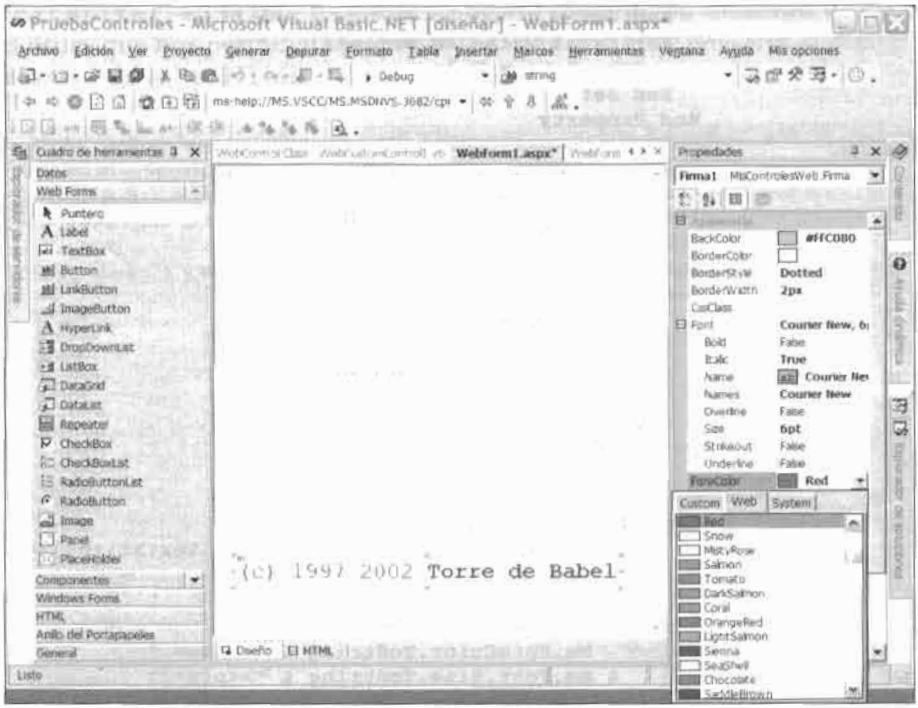


Figura 17.2. Aspecto del control tras modificar el color y tamaño de letra

Suponga que, en ocasiones, quiere preceder la firma que genera el control `Firma` por la fecha e, incluso, la hora actual en el servidor. Estos elementos, no obstante, no aparecerán en todas las páginas, por lo que deberá añadir sendas propiedades que le permitan configurar la aparición o no de esos elementos. Además de las propiedades, como acaba de anotarse, deberá modificar el método redefinido `Render()`. Con todo, nuestra clase de componente podría quedar como sigue:

```
Imports System.ComponentModel
Imports System.Web.UI

Public Class Firma
  Inherits WebControls.WebControl

  Private FShowDate As Boolean
  Private FShowTime As Boolean

  <Category("Apariencia"), _
   Description("Indica si debe mostrarse la fecha"), _
   DefaultValue(False)>
  Public Property ShowDate() As Boolean
    Get
      Return FShowDate
```

```
      End Get
      Set(ByVal Value As Boolean)
        FShowDate = Value
      End Set
    End Property

    <Category("Apariencia"), _
     Description("Indica si debe mostrarse la fecha"), _
     DefaultValue(False)> _
    Public Property ShowTime() As Boolean
      Get
        Return FShowTime
      End Get
      Set(ByVal Value As Boolean)
        FShowTime = Value
      End Set
    End Property

    Protected Overrides Sub Render( _
       ByVal writer As System.Web.UI.HtmlTextWriter)

      writer.Write("<font color=" & _
            Me.ForeColor.ToString & " size= " _
          & Me.Font.Size.ToString & "><pre>")

      If FShowDate Then
        writer.Write(Date.Now.ToLongDateString)
        If FShowTime Then
          writer.Write(" - ")
        Else
          writer.Write("<br>")
        End If
      End If

      If FShowTime Then
        writer.Write(Date.Now.ToLongTimeString & "<br>")
      End If

      writer.Write( _
         "<center>(c) 1997-2002 <b>Torre de Babel</b></center>")
      writer.Write("</pre></font>")
    End Sub
  End Class
```

Tras este cambio, y como se ve en la figura 17.3, podrá activar o desactivar la visualización de la fecha y hora simplemente modificando una propiedad.

La clase `HtmlTextWriter`

Hasta ahora, en los ejemplos previos, tan sólo nos hemos servido del método `Write()` del parámetro recibido, un parámetro que pertenece a una clase,

HtmlTextWriter, en la que tenemos a nuestra disposición multitud de elementos útiles que nos permiten componer la salida que debe enviarse al cliente sin necesidad de saber HTML.

Figura 17.3. El control mostrando la hora y la firma

Lo primero que encontramos en la clase HtmlTextWriter son una serie de constantes que nos facilitan ciertos caracteres especiales que, siendo de uso habitual en la composición de código HTML, no son habituales o resultan difíciles de incluir como literales desde nuestro código Visual Basic. Algunos de esos caracteres son las dobles comillas, los símbolos de apertura y cierre de marca HTML o el carácter de tabulación. Estas constantes son compartidas, lo cual significa que podemos usarlas directamente como HtmlTextWriter.DefaultTabString o HtmlTextWriter.DoubleQuoteChar, por poner un ejemplo.

Para la inclusión de marcas y atributos esta clase dispone de los métodos WriteBeginTag(), WriteEndTag(), WriteAttribute() y WriteStyleAttribute(). Su uso hace innecesaria la codificación manual de marcas como <font> o , usadas en nuestro ejemplo. También existen enumeraciones que representan a la mayoría de marcas más habituales, así como a los atributos de estilo.

Podríamos conseguir exactamente el mismo resultado que ya teníamos con esta nueva versión del método Render():

```
Protected Overrides Sub Render( _
   ByVal writer As System.Web.UI.HtmlTextWriter)
```

```
    writer.WriteBeginTag("font")
    writer.WriteAttribute("color", _
      Drawing.ColorTranslator.ToHtml(Me.ForeColor))
    writer.WriteAttribute("size", Me.Font.Size.ToString)
    writer.WriteEndTag("font")
    writer.WriteFullBeginTag("pre")


    If FShowDate Then
      writer.Write(Date.Now.ToLongDateString)
      If FShowTime Then
        writer.Write(" - ")
      Else
        writer.WriteFullBeginTag("br")
      End If
    End If


    If FShowTime Then
      writer.Write(Date.Now.ToLongTimeString)
      writer.WriteFullBeginTag("br")
    End If


    writer.WriteFullBeginTag("center")
    writer.Write("(c) 1997-2002 ")
    writer.WriteFullBeginTag("b")
    writer.Write("Torre de Babel")
    writer.WriteEndTag("b")
    writer.WriteEndTag("center")
    writer.WriteEndTag("pre")
  End Sub
```

Observe que utilizamos el método `WriteFullBeginTag()` para todas las marcas que no cuentan con atributos, mientras que nos servimos de `WriteBeginTag()` para marcas que, como `font`, van acompañadas de dichos elementos. El código tiene más apariencia de código Visual Basic y menos de código HTML.

Incluso cabría otra versión de `Render()` en la que sustituyésemos las llamadas a `WriteBeginTag()`, `WriteAttribute()` y `WriteEndTag()` por `RenderBeginTag()`, `AddAttribute()` y `RenderEndTag()`, respectivamente, sustituyendo las cadenas `font`, `color` y similares por constantes que son equivalentes y forman parte de enumeraciones predefinidas.

Controles compuestos

Nuestros controles web también pueden estar compuestos de otros controles, en lugar de dibujarse a sí mismos generando código HTML. El desarrollo

de estos controles, no obstante, no es visual sino que se implementa mediante código. A pesar de todo los requerimientos son simples y no requieren demasiado esfuerzo.

Lo primero que hay que hacer es añadir a nuestra clase la implementación de la interfaz `INamingContainer`. Dicha interfaz en realidad no dispone de ningún miembro, sirviendo sólo como una marca para identificar a aquellos controles web que, a su vez, contienen otros. La existencia de esta interfaz en un control pone en marcha un mecanismo de identificación que evitará conflictos entre los controles.

A continuación tenemos que redefinir el método `CreateChildControls()`, insertando en él todo el código que sea preciso para añadir y personalizar nuestros controles. Éstos, finalmente, deberán ser añadidos a la colección de controles representada por la propiedad `Controls`.

Básicamente es como si creáramos los controles de manera dinámica y, por tanto, estableciésemos mediante código sus propiedades y eventos en lugar de hacerlo usando un diseñador.

Veamos con un simple ejemplo cómo podemos crear un control compuesto. Para ello vamos a crear de nuevo el control de conversión entre pesetas y euros que ya desarrollamos en el capítulo anterior, pero en esta ocasión en versión Web.

El código mostrado a continuación seria el que tendríamos que añadir al módulo correspondiente al nuevo control.

```
Imports System
Imports System.web
Imports System.Web.UI
Imports System.Web.UI.WebControls

Public Class EuroCalc
  Inherits WebControl

  Implements INamingContainer


  Private WithEvents Euros As New WebControls.TextBox()
  Private WithEvents Pesetas As New WebControls.TextBox()


  Protected Overrides Sub CreateChildControls()

    Euros.AutoPostBack = True
    Euros.Text = "0"
    Me.Controls.Add(Euros)

    Me.Controls.Add(New LiteralControl("€<br>"))

    Pesetas.AutoPostBack = True
    Pesetas.Text = "0"
    Me.Controls.Add(Pesetas)

    Me.Controls.Add(New LiteralControl("Pts<br>"))
  End Sub
```

```
    Private Sub EurosChanged(ByVal sender As Object, _
        ByVal e As EventArgs) Handles Euros.TextChanged

      Pesetas.Text = Math.Round(Euros.Text * 166.386)
    End Sub


    Private Sub PesetasChanged(ByVal sender As Object, _
        ByVal e As EventArgs) Handles Pesetas.TextChanged

      Euros.Text = Math.Round(Pesetas.Text / 166.386, 2)
    End Sub

  End Class
```

Observe que declaramos los dos controles más importantes, las cajas de texto, con la cláusula `WithEvents`, a fin de poder gestionar sus correspondientes eventos `TextChanged` desde un método a propósito para ello. Hemos dado el valor `True` a la propiedad `AutoPostBack` de ambos controles, de tal forma que cuando se produzca un cambio en su contenido, en el cliente remoto, automáticamente se produzca una comunicación con el servidor para efectuar el cálculo y preparar la respuesta.

Para insertar el contenido estático nos hemos servido de la clase `LiteralControl`, creando objetos con un símbolo de euro de pesetas seguido de una marca de separación.

Por lo demás, no hay nada en el código que no conozcamos ya. Para comprobar el funcionamiento de este control no tiene más que iniciar una nueva aplicación basada en formularios web, insertarlo y ejecutar el proyecto. En la figura 17.4 puede verlo en funcionamiento.

Figura 17.4. El control de conversión en funcionamiento en una página ASP.NET

Pagelets

Al diseñar una sede web completa, en la que existen infinidad de documentos, es habitual que ciertos elementos se repitan con frecuencia. Un ejemplo puede ser la cabecera o el pie de los documentos, donde suele incluirse información acerca de los derechos de autor, enlaces y directorios del lugar. Otro ejemplo podría ser un formulario de entrada de datos que se utilice en varios puntos.

En lugar de introducir el mismo código HTML una y otra vez, para repetir esos elementos, con ASP.NET tenemos la posibilidad de convertir fácilmente dichas porciones de código en componentes. Éstos no son componentes de formularios Web, no aparecerán en el Cuadro de herramientas, sino elementos, conocidos como *pagelets*, que deben registrarse apropiadamente antes de ser usados. El mecanismo es realmente sencillo y el resultado es que tenemos un componente ASP.NET más, aunque definido por nosotros y usando sólo HTML.

Nota

En cierta forma un *pagelet* es como un control compuesto, si bien su desarrollo no se basa en la escritura de código Visual Basic .NET sino HTML. Estos controles no se alojan en bibliotecas compiladas, que pueden distribuirse para su uso en aplicaciones Web, sino que se almacenan como texto y sólo pueden añadirse tras crear una aplicación web ASP.NET

Creación de un *pagelet*

Para ver cómo crearíamos un *pagelet*, nombre con el que se conoce a estos componentes, lo mejor es hacerlo directamente con un ejemplo. Supongamos que estamos creando una sede web para esta misma guía práctica que tiene en sus manos. En las distintas páginas podría verse la portada, obtener datos comerciales, ver el índice de contenidos, etc. La parte repetitiva es que en todas esas páginas nos gustaría incluir una cabecera, y quizá también un pie, en el que aparezca resaltado el título de la guía y un enlace con la sede web de la editorial.

Lo que haríamos sería añadir al proyecto un Control de usuario Web (véase figura 17.5) insertando en él el código HTML para definir la apariencia visual de esa cabecera. Este archivo lo guardaríamos asignándole la extensión `ascx` (*ASP Component*) y, en este ejemplo concreto, llamaríamos al archivo `Cabecera.ascx`. Su contenido sería el siguiente:

```
<p>
 <font face="verdana" size="3"><font>
  <table bgcolor="yellow" width="100%">
   <td>
    Programación con Visual Basic .NET
```

```
    </td>
    <td>
      <a href="http://www.anayamultimedia.es">
        <b>Anaya Multimedia</b>
      </a>
    </td>
  </table>
</p>
```

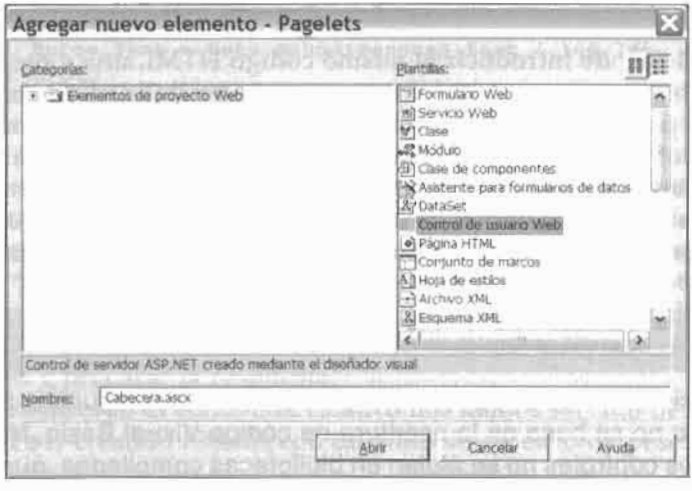


Figura 17.5. Añadimos al proyecto un control Web de usuario

Observe que no hay nada especial en este código, tan sólo HTML definiendo una tabla con dos celdas y unos ciertos atributos. Aunque no lo hayamos hecho, en la práctica también es posible usar componentes de servidor ASP.NET en un *pagelet*.

Nota

Al crear el control de usuario, en principio se encontrará con una superficie de diseño idéntica a la de un formulario Web. Acceda a la página de código HTML para añadir el contenido anterior.

Uso de un *pagelet*

Lo que tenemos en este momento en el archivo `Cabecera.ascx` es, simplemente, un fragmento de código HTML, nada más. Para convertir ese fragmento en un componente ASP.NET tendremos que usar la directiva `<%@ Register` de ASP.NET facilitando como parámetros un prefijo y un nombre para el componente, así como el nombre del archivo en que se encuentra el código. Esto lo haríamos al inicio de cada documento ASP.NET donde vayamos a utilizar el *pagelet*.

Un ejemplo sería la línea siguiente añadida al principio del módulo `WebForm1.aspx` de cualquier proyecto web:

```
<%@ Register TagPrefix="PControl"
    TagName="Cabecera" Src="Cabecera.ascx" %>
```

En este caso hemos asignado al componente el prefijo `PControl` y el nombre `Cabecera`. Esto significa que a la hora de incluirlo en nuestro documento ASP.NET usaríamos la referencia `PControl:Cabecera`, de forma análoga a como usamos la referencia `asp:Calendar` para incluir un calendario ASP.NET.

Nos bastaría con añadir el *pagelet* donde nos interese, por ejemplo antes y después del formulario del ejemplo anterior, usando la siguiente línea:

```
<PControl:Cabecera runat="server"
    ID="Cabecera2" NAME="Cabecera2" />
```

El diseñador de formularios Web mostrará el código introducido, haciendo referencia al componente, tal como se muestra en la figura 17.6. El componente aparece como un rectángulo en el que se indica que es un control de usuario. Podemos colocarlo donde nos interese. Al ejecutar el proyecto, ese componente generará la cabecera y el pie que puede ver en la figura 17.7. Nos ahorramos bastante trabajo, especialmente si dicha cabecera y pie deben incluirse en todos los documentos de nuestra sede web.

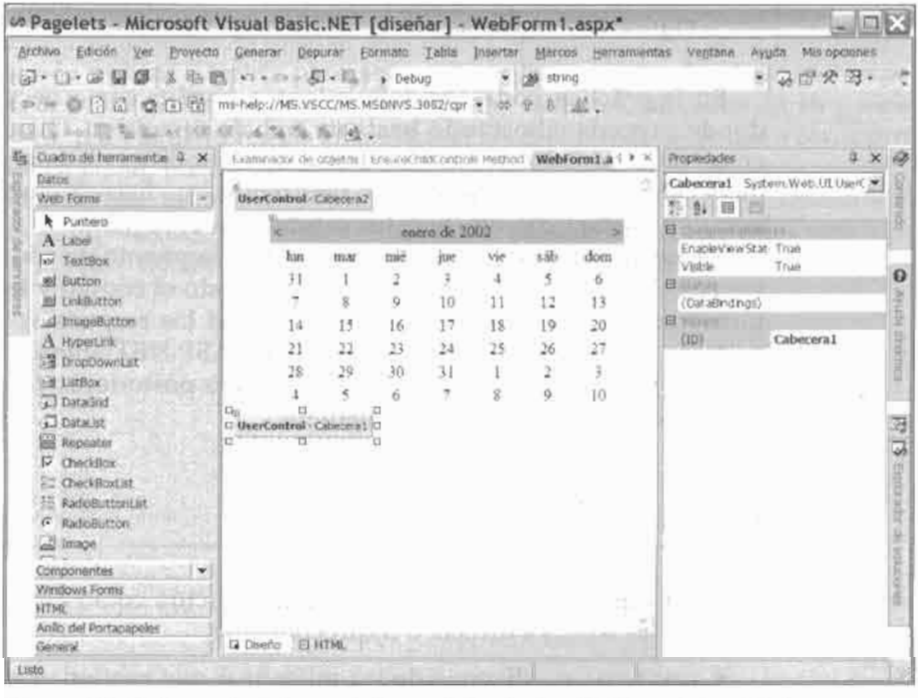


Figura 17.6. Aspecto del *pagelet* en el diseñador de formularios Web

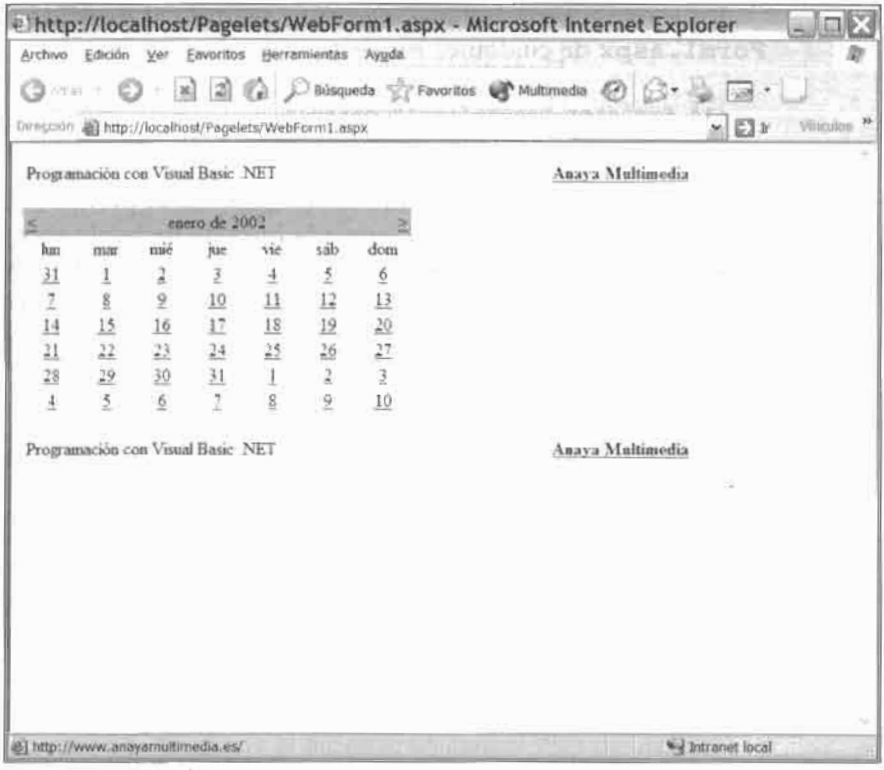


Figura 17.7. Resultado de la ejecución del proyecto

En la práctica podemos definir tantos *pagelets* como necesitemos y usarlos donde proceda, ahorrando bastante trabajo repetitivo. Otra ventaja es que si, en un determinado momento, optamos por modificar la apariencia o contenido de la cabecera de las páginas, no tendremos que hacerlo documento por documento y bastará con efectuar los cambios en el *pagelet*.

Recuerde que, aunque el *pagelet* sea un fragmento de código HTML, el motor ASP.NET efectúa una compilación de todo el código y lo almacena en una memoria intermedia o *cache*, acelerando así las respuestas al cliente. Sólo la primera vez que se solicite el documento ASP.NET anterior, por tanto, se incluye el *pagelet* y analiza el código. Las veces posteriores ya existe una versión compilada y la respuesta es inmediata.

Puntos clave

- La clase `System.Web.UI.Control` actúa como base de todos los controles web, ya directa o indirectamente.
- `Control` no dispone de los miembros que gestionan el posicionamiento y apariencia de un control, por lo que si nos interesan esos elementos

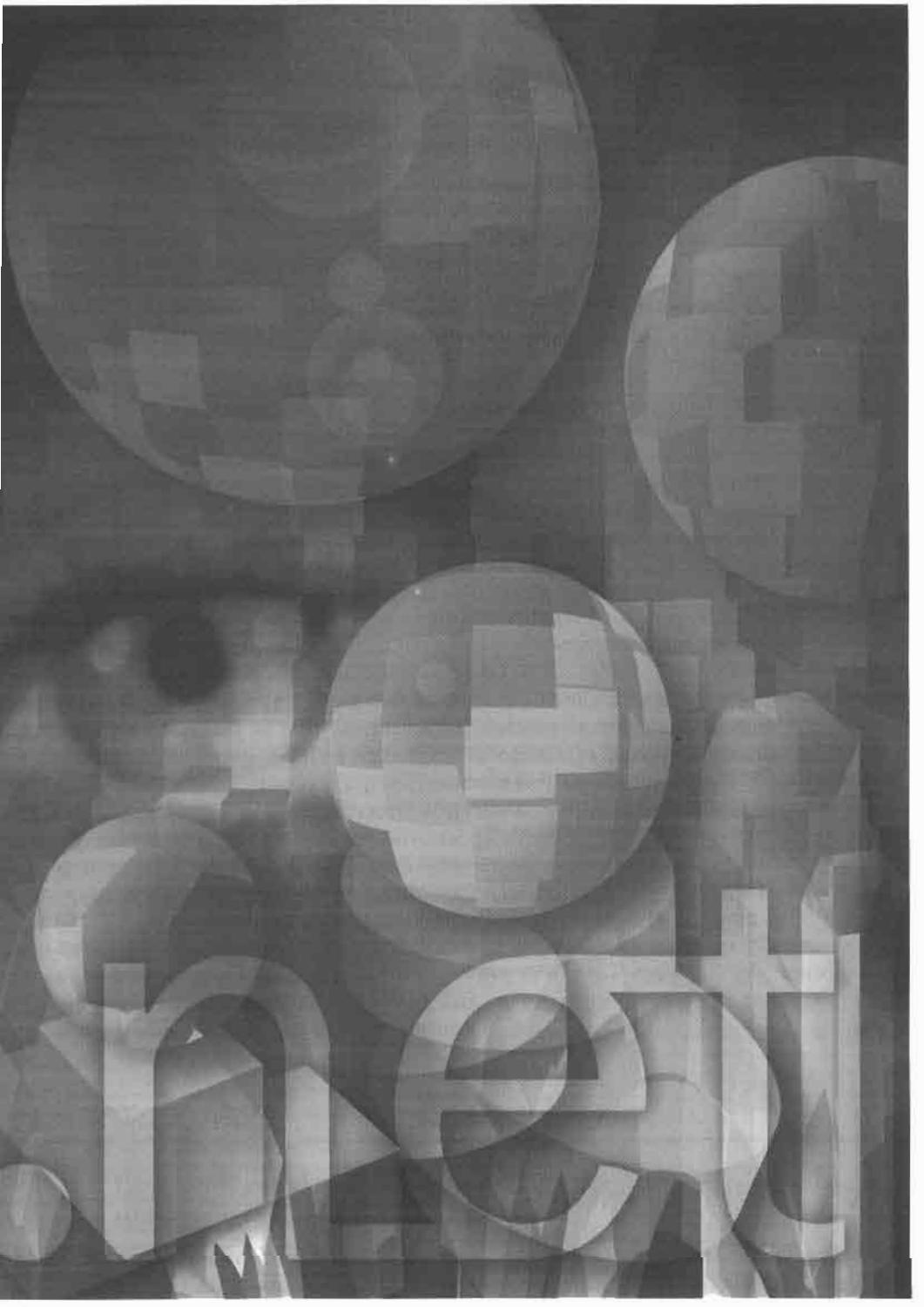
tomaremos como base a `System.Web.UI.WebControls.WebControl` en lugar de partir directamente de `Control`.

- Los controles web que no contienen a otros generan su apariencia en el formulario, la página web que después se envía al cliente, redefiniendo la implementación por defecto del método `Render()`.
- Si el control está compuesto de otros el método a redefinir es `CreateChildControls()`, creando todos los controles hijo y añadiéndolos a la colección `Controls`.
- Usando las propiedades y métodos de la clase `HtmlTextWriter` no es necesario introducir manualmente todas las marcas y atributos HTML para generar una cierta apariencia, ya que parte del trabajo está automatizado.
- También podemos crear controles compuestos de otros elementos utilizando sólo y exclusivamente HTML. A estos controles se les conoce como *pagelets* o controles de usuario web.

Resumen

Lo aprendido en este capítulo le servirá como base para comenzar a desarrollar sus propios controles web, campo amplísimo sobre el cual podrían escribirse cientos de páginas. Lo fundamental es que sabe, básicamente, los recursos que tiene a su alcance. Al diseñar una nueva aplicación basada en formularios web puede utilizar *pagelets* para automatizar la inserción de elementos repetitivos, dejando el diseño de controles independientes, alojados en su propia biblioteca, para aquellos casos en los que el control pueda serle útil a otros proyectos o, incluso, pueda servir a terceros.

Recuerde que todo lo aprendido en capítulos previos sobre orientación a objetos, componentes y atributos es aplicable también a los controles web que, al fin y al cabo, no son más que clases de objetos especializados.



18 Creación de servicios Windows

No todas las aplicaciones que se ejecutan en Windows lo hacen de manera interactiva, siendo iniciadas y controladas por un usuario como es lo habitual en los tipos de proyecto desarrollados en capítulos previos. Existen aplicaciones que no cuentan con una interfaz de usuario, es más, no necesitan que se inicie una sesión para efectuar su trabajo que, por regla general, queda en segundo plano sin interferir con el uso cotidiano que pueda hacerse del sistema.

La creación de servicios Windows requiere que el programador siga unas ciertas reglas en su desarrollo, reglas que no era posible alcanzar con versiones previas de Visual Basic, de forma que la creación de este tipo de proyectos quedaba fuera de nuestras posibilidades, siendo necesario recurrir a otras herramientas. Visual Basic .NET, por el contrario, no sólo tiene la capacidad necesaria para crear servicios Windows sino que, además, cuenta con unos asistentes que realizan gran parte del trabajo, simplificando considerablemente el nuestro.

Nuestro objetivo, en este capítulo, es conocer las clases que están implicadas en el desarrollo de servicios Windows, su instalación y desinstalación y su control: inicio, pausa, detención, etc. Nos serviremos de un ejemplo simple pero demostrativo que, además, nos servirá también para conocer nuevos servicios de la plataforma .NET, concretamente los relacionados con la comunicación mediante *sockets*.

Esquema general

Antes de ponernos manos a la obra, conociendo con detalle las clases que necesitamos y poniéndolas en práctica para desarrollar nuestro primer servicio, vamos a hacer un recorrido general por el proceso de desarrollo, instalación y control de un servicio, una primera aproximación que nos permita tener una visión global sobre este tema.

Podemos implementar como servicio Windows cualquier aplicación que, ejecutándose sobre dicho sistema operativo, no necesite la intervención de ningún usuario. Los servicios desempeñan tareas que quedan en segundo plano pero que resultan imprescindibles en muchos casos. SQL Server, por poner un ejemplo, es una aplicación que se ejecuta como si fuera un servicio de tal forma que cualquier programa, sin actuación explícita sobre SQL Server, puede acceder a una base de datos y trabajar con la información que contiene. IIS (*Internet Information Server*), el servidor web incorporado en Windows NT/2000/XP, es otro ejemplo de aplicación que funciona como servicio Windows.

Para conocer y controlar el estado de los servicios podemos usar diversas herramientas de Windows. La más habitual es la utilidad Servicios que aparece en el Panel de control y que, en el caso de Windows XP, tiene la apariencia que puede verse en la figura 18.1. En la lista aparecen todos los servicios instalados en el sistema, indicándose su estado y cómo se inician. Al seleccionar un servicio cualquiera, en la parte izquierda aparecen las opciones disponibles para iniciarlo, detenerlo, pausarlo o reanudarlo.

Otra posibilidad es recurrir a la consola y utilizar el comando `NET START` para ver los servicios que hay en marcha. Este mismo comando, seguido del nombre de un servicio, lo inicia, mientras que `NET STOP` y el nombre de un servicio lo detiene.

En la plataforma .NET un servicio es cualquier aplicación que pone en ejecución una clase derivada de `System.ServiceProcess.ServiceBase`. Dicha clase cuenta con los elementos apropiados para que utilidades como las que acaban de describirse puedan interactuar con el servicio, permitiendo su inicio y detención. En el punto siguiente comenzaremos a estudiar esta clase y veremos cómo utilizarla para construir nuestro servicio.

También en el ámbito `System.ServiceProcess` encontramos las clases `ServiceInstaller` y `ServiceController`. La primera se utiliza para crear el instalador con que debe contar cada servicio, mientras que mediante la segunda es posible desarrollar utilidades para controlar servicios, al estilo de Servicios del Panel de control.

Tras desarrollar un servicio, creando una nueva clase derivada de `ServiceBase`, obtenemos un módulo que no puede ejecutarse directamente. Es necesario instalarlo para que aparezca en la lista de servicios del sistema y que sea el usuario quien lo inicie o detenga a demanda. Para efectuar esa instalación utilizaremos la herramienta `installutil` que acompaña a Visual Basic .NET, herramienta que buscará una clase derivada de `System.Configuration.Install.Installer` para personalizar la instalación. La misma utilidad nos servirá en caso de que deseemos desinstalar el servicio.

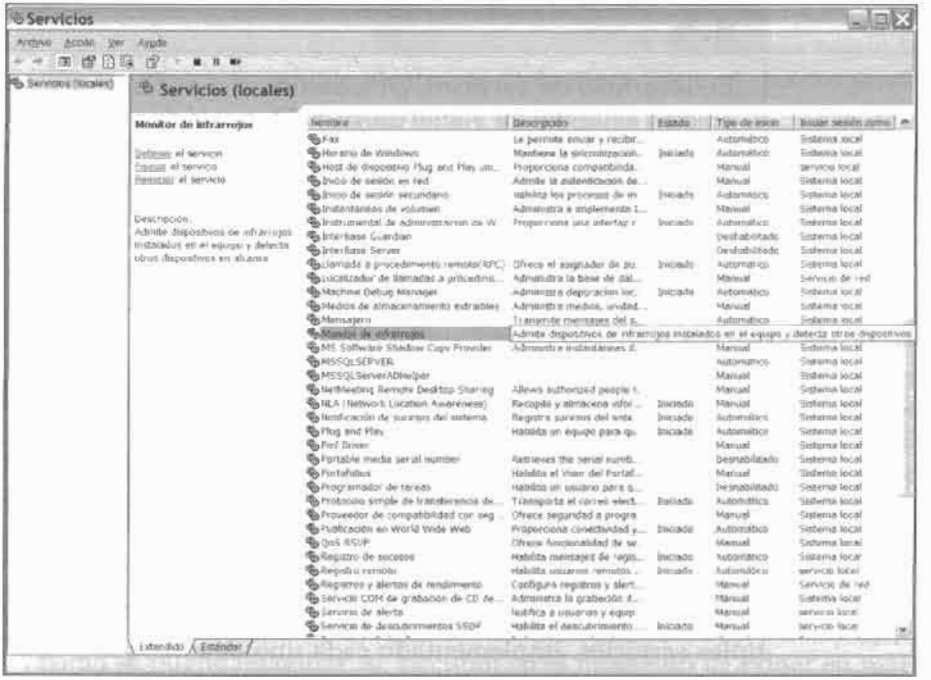


Figura 18.1. Utilidad de gestión de servicios de Windows XP

Figura 18.2. Uso del comando `NET START` desde la consola del sistema

Nota

El desarrollo de servicios Windows es una posibilidad que no existe en la edición estándar de Visual Basic .NET, por lo que si dispone de esa edición no encontrará los asistentes que van a mencionarse en el punto siguiente y seguramente tampoco encuentre la herramienta `installutil`.

La clase `ServiceBase`

Para que nuestra aplicación pueda considerarse un servicio deberá contar con una clase que esté, directa o indirectamente, derivada de `ServiceBase`, una clase que, como ya sabe, está alojada en el ámbito `System.ServiceProcess`. Derivada directa de la clase `Component`, que conocemos de capítulos previos, esta clase añade un conjunto de propiedades que determinan las operaciones de control que el usuario podrá ejecutar sobre el servicio, así como una serie de métodos que se ejecutarán cuando esas operaciones tengan lugar.

La propiedad `ServiceName` contendrá el nombre con el que se identificará el servicio, teniendo en cuenta que en un mismo proyecto podemos tener múltiples servicios, implementado cada uno de ellos como una clase derivada de `ServiceBase`. Las propiedades `CanStop`, `CanShutdown` y `CanPauseAndContinue` indican si el servicio puede detenerse, liberarse, pausarse y continuarse, respectivamente. Sólo la primera tiene, inicialmente, el valor `True`.

Por regla general, las aplicaciones corrientes son cerradas por el propio usuario, o automáticamente, cuando el sistema se encuentra en ciertas circunstancias, como puede ser la ausencia de suministro eléctrico. Un servicio indicará, mediante la propiedad `CanHandlePowerEvent`, si está preparado o no para responder a estas circunstancias, por ejemplo deteniendo su ejecución o pasando a algún otro estado.

Creación del objeto que actúa como servicio

La aplicación que va a actuar como servicio deberá contar con un método compartido llamado `Main()`, como prácticamente cualquier otro tipo de aplicación. Desde ese método invocaremos al método `Run()` de `ServiceBase` facilitando como parámetro un objeto de la clase derivada de `ServiceBase`, de manera análoga a como pasamos un objeto derivado de `Form` al método `Run()` de la clase `Application`. El esqueleto básico sería el siguiente:

```
Public Class MiClaseServicio
  Inherits System.ServiceProcess.ServiceBase

  Shared Sub Main()
    System.ServiceProcess.ServiceBase.Run(new MiClaseServicio)
  End Sub
...
```

En este caso usamos la versión del método `Run()` que acepta como parámetro sólo un objeto de una clase derivada de `ServiceBase`. Existe una segunda versión que acepta un arreglo de objetos de dicho tipo, iniciando varios servicios de una sola vez.

Lo que hacemos en el método `Main()` es preparar el servicio, alojando su ensamblado en memoria y dejándolo preparado para que, desde la línea de comandos o mediante la utilidad Servicios ya mencionada, pueda ser puesto en marcha, detenido, etc.

Eventos operacionales

Una vez que el servicio está alojado en memoria, su funcionamiento vendrá determinado por varios eventos de la clase `ServiceBase`. En realidad esos eventos aparecen como métodos protegidos redefinibles en nuestra clase, siguiendo el esquema que vimos en los capítulos previos al tratar el desarrollo de componentes. Si queremos, por ejemplo, responder al evento que se produce cuando el usuario solicita pausar un servicio, no tenemos más que redefinir el método protegido `OnPause()`.

Como mínimo, si no hemos modificado las propiedades que inicialmente tiene el servicio, deberemos redefinir los métodos `OnStart()` y `OnStop()` que, tal como se puede imaginar, se ejecutan en el momento de poner en marcha y detener el servicio. También podemos redefinir los métodos `OnPause()`, `OnContinue()`, `OnShutdown()` y `OnPowerEvent()` en caso de que hayamos dado el valor `True` a las propiedades correspondientes.

Cuando se invoque a `OnStart()` es que el servicio debe ponerse en marcha. Puede efectuar toda su acción en dicho método o poner en marcha un hilo de ejecución independiente, según las necesidades. El servicio no puede ponerse más de una vez en marcha, pero si efectuamos un proceso muy largo en el evento `OnStart()` debemos habilitar algún mecanismo que permita detenerlo en caso de que se ejecute `OnStop()`.

Instalación del servicio

Para que nuestro servicio, desarrollado según las directrices que acaban de darse, aparezca en la lista de servicios disponibles en el sistema es necesario instalarlo y, con este fin, deberemos incluir en el mismo ensamblado una clase que tenga el atributo `RunInstaller` y facilite el proceso necesario para incluir en el registro de Windows las claves y valores apropiados.

En el ámbito `System.Configuration.Install` existe una clase, llamada `Installer`, que, derivada de `Component`, sirve como base de los distintos tipos de instaladores que podemos necesitar en una aplicación .NET. Dos de las derivadas de dicha clase son `ServiceInstaller` y `ServiceProcessInstaller`. La primera de ellas se encarga de la instalación de los servicios, mientras que la segunda se encarga de la configuración del ensamblado en el que se encuentran dichos servicios.

La utilidad de instalación `installutil`, mencionada anteriormente, buscará en el ensamblado que contiene nuestro servicio una clase que tenga el atributo `RunInstaller`. Dicha clase, además, deberá estar derivada de `Installer` y, básicamente, tendrá que añadir a la colección `Installers` los objetos que se ocuparán del proceso de instalación. `Installutil` recuperará esos objetos de la mencionada colección e invocará a los métodos `Install()` de cada uno de ellos, completando la tarea.

Por cada servicio que exista en nuestro proyecto crearemos un objeto de la clase `ServiceInstaller`, asignando los valores apropiados a `ServiceName` y `StartType`. La primera tendrá el nombre del servicio, mientras que la segunda determinará el modo en que éste será iniciado. Los valores posibles de esta segunda propiedad son `Automatic`, `Disabled` y `Manual`, según deseemos que el servicio se inicie automáticamente al poner en marcha el sistema, esté inaccesible o tenga que ser iniciado manualmente.

Nota

Es importante que el valor que asignemos a la propiedad `ServiceName` del objeto `ServiceInstaller` coincida con el nombre que hemos dado previamente al servicio en la propiedad `ServiceName` de nuestra clase derivada de `ServiceBase`, de lo contrario la instalación fallará.

Además de un objeto `ServiceInstaller`, nuestra clase de instalación también debe crear un objeto de la clase `ServiceProcessInstaller`. Éste se encargará de la configuración no de los servicios sino del ensamblado en que están alojados determinando, por ejemplo, la cuenta que utilizarán para ejecutarse y, si es necesario, el nombre y la clave de usuario.

Asistentes disponibles para desarrollar servicios

Como en muchos otros casos, podríamos desarrollar completamente un servicio Windows sin necesidad de usar entorno alguno, tan sólo escribiendo código. La ventaja del entorno de desarrollo de Visual Basic .NET, sin embargo, es que dispone de asistentes y diseñadores capaces de facilitar este trabajo generando parte de ese código. En este caso concreto los asistentes, o plantillas de código, son dos. Para iniciar el desarrollo de un servicio Windows seleccionaremos el elemento Servicio de Windows de la ventana Nuevo proyecto, como se aprecia en la figura 18.3. Aparecerá un diseñador en el que podremos insertar componentes que pudiéramos necesitar para nuestro servicio, por ejemplo un `MessageQueue` o un `FileSystemWatcher`. Estos elementos dependerán de la propia funcionalidad del servicio.

Cambiando al editor verá que ya tenemos una clase derivada de `ServiceBase`. Ésta cuenta con el método compartido `Main()` y los métodos `OnStart()` y `OnStop()`. En `Main()` encontramos el código siguiente:

```
Shared Sub Main()
  Dim ServicesToRun() As System.ServiceProcess.ServiceBase

  ServicesToRun = New System.ServiceProcess.ServiceBase() _
      {New ServiceHorario()}

  System.ServiceProcess.ServiceBase.Run(ServicesToRun)
End Sub
```

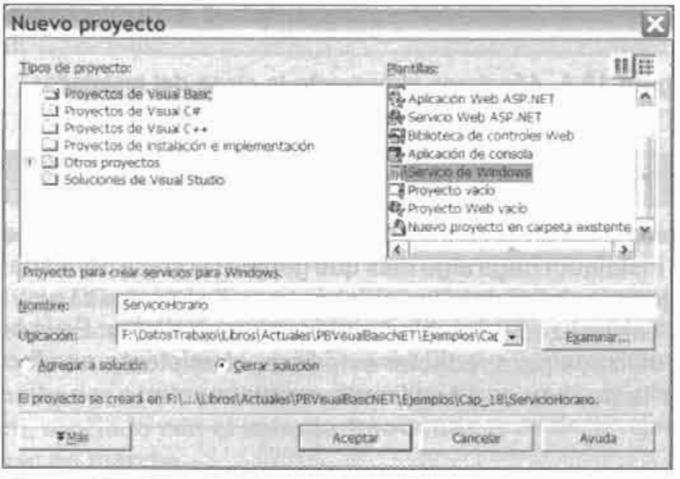


Figura 18.3. Tenemos un elemento que genera el esqueleto del servicio

Lo que se hace es definir un arreglo de objetos `ServiceBase` que, en este caso concreto, tan sólo cuenta con un elemento. Ese arreglo es facilitado como parámetro al método `Run()` de la clase `ServiceBase`, tal y como se indicaba en un párrafo anterior. A la vista del esqueleto generado por el asistente, lo único que tenemos que hacer nosotros es añadir el código de implementación a los métodos `OnStart()` y `OnStop()`. Como se indica en los propios comentarios introducidos por el asistente, en caso de definir más servicios tendremos que modificar el método `Main()` añadiendo los elementos necesarios al arreglo, antes de la llamada al método `Run()`.

Finalizada la implementación de los diversos servicios que vayamos a programar, el paso siguiente será añadir al proyecto una nueva Clase del instalador, usando para ello la ventana Agregar nuevo elemento (véase figura 18.4). Obtendremos un nuevo módulo de código con una clase derivada de `Installer`, tal y como se apuntaba antes, en cuyo constructor crearemos los instaladores correspondientes al servicio. Con esto ya tendriamos finalizado el desarrollo de la solución.

Figura 18.4. Añadimos al proyecto la clase del instalador

Nota

Posiblemente en la versión definitiva de Visual Basic .NET el asistente Clase de instalador haga algo más que generar la clase derivada de `Installer`, por ejemplo definiendo los objetos `ServiceInstaller` y `ServiceProcessInstaller` y estableciendo sus propiedades. En la *beta 2*, que ha sido la utilizada para redactar este libro, el asistente no efectúa ese trabajo y, por tanto, hay que codificarlo manualmente.

Un servicio horario para red

Hasta ahora hemos conocido teóricamente todos los elementos implicados en el funcionamiento y desarrollo de un servicio Windows, desde la utilidad de control y el comando `NET` hasta las clases `ServiceBase`, `ServiceInstaller` y `ServiceProcessInstaller`. En este punto vamos a poner ese conocimiento en práctica y, para ello, desarrollaremos un servicio Windows que, mientras está en funcionamiento, se mantendrá a la escucha en un cierto puerto TCP/IP, facilitando la fecha local del sistema a cualquier equipo que pudiera solicitarlo a través de una red local o Internet.

Para desarrollar este proyecto usaremos los servicios de comunicación TCP/IP de la plataforma .NET, también conocidos como servicios de *sockets* o conectadores. Dichos servicios, representados como siempre por distintas clases, se alojan en el ámbito `System.Net.Sockets`.

El servicio pondrá un hilo de ejecución en marcha cuando se produzca el *evento* `OnStart()`, *hilo en el que se quedará a la espera de las peticiones de* los clientes. Veamos cuáles serían los pasos que tendríamos que dar.

Definición de la clase de servicio

Usaremos el asistente anteriormente mencionado para generar el esqueleto del servicio, modificando a continuación la clase. Lo primero será añadir al inicio del módulo las siguientes sentencias `Imports`, necesarias para poder acceder cómodamente a los servicios de *sockets*, *threads* y conversión de texto.

```
Imports System.Net.Sockets
Imports System.Threading
Imports System.Text
```

No es necesario alterar el constructor de la clase ni el método `Main()`, ya que ambos efectúan su trabajo apropiadamente. Si añadiremos a la clase dos miembros nuevos:

```
Private MiSocket As New TcpListener(1010)

Private MiThread As New Thread(AddressOf Escucha)
```

La variable `MiSocket` es un objeto `TcpListener` que nos permitirá *escuchar* en nuestro sistema por un cierto puerto de comunicación, en este caso el `1010`. Asumimos que ese canal estará libre y podemos modificarlo libremente por cualquier otro en caso necesario. `MiThread` es un objeto `Thread`, que ya conoce, asociado con el método `Escucha()`, es decir, ejecutará ese método al ponerse en marcha.

A continuación modificaremos los métodos `OnStart()` y `OnStop()`, cuyos prototipos han sido introducidos ya en el módulo por el asistente. La implementación, como puede verse a continuación, es realmente sencilla:

```
Protected Overrides Sub OnStart(ByVal args() As String)
  MiSocket.Start()
  MiThread.Start()
End Sub

Protected Overrides Sub OnStop()
  MiSocket.Stop()
End Sub
```

Traducimos la puesta en marcha del servicio en una llamada al método `Start()` del `TcpListener` y del `Thread`. La primera abre el canal de comunicación para poder utilizarlo, mientras que la segunda inicia el hilo que se encargará de ejecutar el método `Escucha()`. La detención del servicio se limita a cerrar el canal de comunicación lo cual, tal y como vamos a ver a continuación,

provocará una excepción en el método `Escucha()` que procederá a finalizar la ejecución del hilo.

Tan sólo nos queda codificar el método `Escucha()` que es, precisamente, el más interesante ya que en él se desarrolla el trabajo del servicio. El código es:

```
Private Sub Escucha()

  Dim Codificacion As New ASCIIEncoding()
  Do While True
    Try

      Dim SocketCliente As Socket = MiSocket.AcceptSocket()

      SocketCliente.Send(Codificacion.GetBytes( _
         DateTime.Now.ToLongDateString.ToCharArray()))
      SocketCliente.Close()
    Catch
      Return
    End Try
  Loop
End Sub
```

Al llegar al método `Escucha()` ya tenemos el `TcpListener` con el canal de comunicación abierto y a la escucha. Usamos su método `AcceptSocket()` para recuperar la petición del cliente, quedando a la espera en caso de que no existiese ninguna. Ese método devolverá un objeto de la clase `Socket` que representa el conducto por el que podemos comunicarnos con el cliente, conducto del que podemos recuperar y por el que podemos enviar información. El método `Send()` es el que nos interesa en este caso, con el objetivo de enviar al cliente la fecha actual sin esperar ningún dato más por su parte.

El método `Send()` necesita como parámetro un arreglo de tipo `Byte` con los bytes a enviar a través del *socket*. Lo que nosotros tenemos es una cadena de caracteres obtenida mediante el método `ToLongDateString()`, conteniendo la fecha. Convertimos esa cadena de caracteres en un arreglo de caracteres sirviéndonos del método `ToCharArray()` de la clase `String`. No es posible entregar un arreglo de tipo `Char` a un método que espera un arreglo de tipo `Byte`, ya que en Visual Basic .NET un `Char` es un carácter UNICODE que ocupa 16 bits, mientras que un `Byte` son 8 bits. Dado que en nuestra lengua los caracteres alfabéticos están todos comprendidos en el conjunto ASCII, nos servimos de un objeto `System.Text.ASCIIEncoding` para, mediante el método `GetBytes()`, obtener el código numérico de cada uno de los caracteres, generando así un arreglo de tipo `Byte` que es lo que necesitamos.

Nota

Revise la documentación del ámbito `System.Text` para obtener más información acerca de `ASCIIEncoding` y su utilidad.

Tras enviar la información al cliente cerramos el canal de comunicación de manera inmediata. Observe que el proceso que acaba de describirse está en el interior de un bucle que se ejecuta sin fin, finalizando la respuesta a un cliente y quedando a la espera de la siguiente solicitud. Esta repetición será constante mientras no se produzca una excepción al llamar a `AcceptSocket()`, caso en el que salimos del método `Escucha()` y, por lo tanto, termina la ejecución del hilo que se había puesto en marcha. ¿Cuándo se generará dicha excepción? En el momento en que desde el método `OnStop()` sea cerrado el canal de comunicación.

Preparación del instalador

Para completar nuestro proyecto tendremos que añadirle el instalador, usando para eso el asistente mencionado en un punto previo. Con esto ya tendremos nuestra clase, derivada de `System.Configuration.Install.Installer`, y tan sólo tendremos que añadir la declaración de dos variables y unas sentencias para configurarlas y añadirlas a la colección `Installer`. La declaración es la siguiente:

```
' Declaramos los dos objetos que necesitamos
Dim SI As New ServiceInstaller()
Dim SPI As New ServiceProcessInstaller()
```

La primera variable es el instalador para el servicio, un objeto que debemos personalizar indicando el nombre del servicio y el método de inicio. Las dos sentencias, que añadiremos al constructor de la clase, son éstas:

```
' Establecemos el nombre del servicio
SI.ServiceName = "ServicioHora"
' Establecemos el modo de inicio
SI.StartType = ServiceStartMode.Manual
```

El valor asignado a `ServiceName` debe ser el mismo que contiene la propiedad `ServiceName` del servicio. Puede establecerlo abriendo la clase creada en el punto anterior en su diseñador y sirviéndose de la ventana Propiedades. El modo de inicio fijado es `Manual`, por lo que el servicio deberá ponerse en marcha manualmente.

También tenemos que configurar el instalador del ensamblado, indicando la cuenta que debe utilizarse para ejecutarlo. Tenemos dos opciones: facilitar un nombre de usuario y una clave, si es que nos interesa que se ejecute con los privilegios de dicho usuario, o bien utilizar las constantes de la enumeración `ServiceAccount` para usar cuentas específicas para servicios. Nosotros optaremos por introducir la sentencia mostrada a continuación, usando la constante `LocalService` que equivale a indicar que debe ejecutarse el servicio en una cuenta con privilegios como para identificarse ante otros sistemas.

```
SPI.Account = ServiceAccount.LocalService
```

El último paso será añadir los dos objetos, `SI` y `SPI`, a la colección de instaladores devuelta por la propiedad `Installers`:

```
Installers.Add(SI)
Installers.Add(SPI)
```

Instalación del servicio

Hemos finalizado el desarrollo de nuestro servicio que, al ser compilado, generará un módulo ejecutable con un ensamblado. Tendremos que abrir la consola del sistema o línea de comandos para proceder a su instalación. No tenemos más que llamar a la utilidad `installutil` facilitando como parámetro el nombre del módulo, incluida la extensión. Esto es lo que se ha hecho en la figura 18.5 que, como se puede apreciar en la parte inferior, ha finalizado satisfactoriamente.

Figura 18.5. Instalamos el servicio en el sistema

Si todo ha ido bien, el servicio deberá aparecer en la lista de servicios de la figura 18.1, punto desde el que podremos ponerlo en marcha y detenerlo. Tam

bién podemos utilizar el comando `NET START ServicioHora` y `NET STOP ServicioHora` para conseguir el mismo resultado desde la línea de comandos.

Comprobación del servicio

Tenemos nuestro servicio instalado e, incluso, en funcionamiento si lo hemos puesto en marcha. No obstante, no sabremos realmente si efectúa su trabajo mientras no lo comprobemos. Podríamos desarrollar una aplicación Windows, un componente o cualquier otro proyecto que, utilizando los servicios de *sockets* de la plataforma .NET conectase con el sistema a través del canal 1010 y mostrase la respuesta obtenida. Sin embargo es un trabajo que no necesitamos ya que basta usar el cliente *telnet* que incluye Windows para ver esa respuesta.

Abra la consola del sistema e introduzca en la línea de comandos la orden `telnet localhost 1010`, con la que indicamos que se inicie el programa `telnet` para conectar con nuestro propio sistema a través del puerto `1010`. De inmediato verá aparecer la fecha y un mensaje indicando que se ha perdido la conexión, lo cual es lógico ya que nuestro servicio cerraba el canal en cuanto enviaba la información, sin esperar ningún retorno.

Como se aprecia en la figura 18.6, el servicio está en funcionamiento y responde adecuadamente a las solicitudes. Pruebe a detenerlo y a conectar, para ver que no obtiene respuesta. También puede modificar la implementación del servicio para que en lugar de la fecha devuelva cualquier otra información útil.

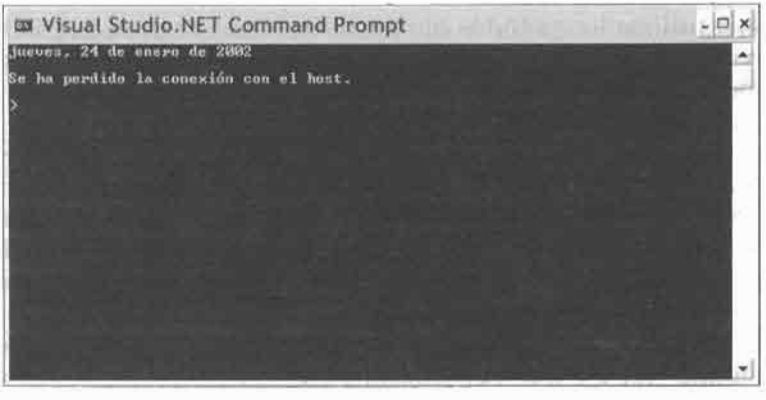


Figura 18.6. Comprobamos que el servicio está funcionando

Controlando los servicios

Ahora que ya sabemos cómo desarrollar un servicio para Windows, utilizando los asistentes y clases descritas en los puntos previos, vamos a dedicar el resto del capítulo a la creación de un controlador para esos servicios. Ya ha visto que nuestro servicio puede controlarse mediante una consola del sistema y un comando, de tal forma que podemos iniciarlo y detenerlo. Esas mismas

operaciones podemos efectuarlas desde una aplicación propia, una aplicación hecha a medida para controlar nuestro servicio.

En realidad podemos controlar cualquier servicio, como veremos en un momento, si bien lo habitual es desarrollar un controlador para un servicio específico, por ejemplo disponiendo un icono en el área de notificación para facilitar el inicio, detención y configuración del servicio. Este último apartado, la configuración, es quizá la más importante, ya que desde la consola de servicios de Windows tan sólo pueden efectuarse tareas estándar, homogéneas para todos los servicios.

La clase `ServiceController`

Para poder conocer el estado y controlar las operaciones sobre un servicio tendremos que crear un objeto de la clase `ServiceController`. Observe que en este caso no hay que derivar una nueva clase, como ocurre al crear al servicio, sino simplemente usar la clase `ServiceController` para acceder al servicio, de igual forma que utilizamos `Thread` para gestionar un hilo de ejecución, por poner un ejemplo.

Esta clase dispone de múltiples constructores sobrecargados, siendo quizá el más interesante el que toma como único parámetro una cadena de caracteres con el nombre del servicio que deseamos controlar. Este nombre es el mismo asignado a `ServiceName` en el momento de desarrollar el servicio que, en nuestro caso, sería `ServicioHora`. Si desconocemos el nombre del servicio, podemos utilizar los métodos compartidos `GetServices()` y `GetDevices()` para recuperar sendos arreglos de objetos `ServiceController` representando a todos los servicios que no están relacionados con dispositivos, en el primer caso, o que sí lo están, en el segundo.

Disponiendo de un objeto `ServiceController` asociado a un servicio, nos serviremos de las propiedades `CanStop`, `CanShutdown` y `CanPauseAndContinue` para conocer las operaciones que es posible efectuar sobre él. También podemos recuperar datos como la descripción del servicio: `DisplayName`; el nombre del ordenador en el que se ejecuta: `MachineName`, o su estado: `Status`. Esta última propiedad puede tener uno de los valores enumerados en la tabla 18.1. Con ellos conoceremos si el servicio está en funcionamiento, detenido, en espera de ser pausado, etc.

Tabla 18.1. Valores posibles en la propiedad `Status`

Constante	Indica que el servicio está
`Running`	En funcionamiento
`StartPending`	Iniciándose
`Stopped`	Detenido
`StopPending`	Deteniéndose
`Paused`	En pausa

Constante	Indica que el servicio está
`PausePending`	En espera de ser pausado
`ContinuePending`	En espera de ser reanudado

Nota

Propiedades como `Status` puede no tener un valor correcto si ha transcurrido cierto tiempo desde que se creó el objeto `ServiceController`, ya que el estado del servicio ha podido cambiar mientras tanto. Siempre podemos llamar al método `Refresh()` antes de acceder a esas propiedades para estar seguros de que el valor obtenido es real.

Para actuar sobre el servicio nos serviremos de los métodos `Start()`, `Stop()`, `Pause()` y `Continue()` que, como puede suponer, inician el servicio, lo detienen, lo pausan o reanudan, respectivamente. Estos son métodos genéricos que representan acciones comunes a todos los servicios.

Si estamos construyendo un controlador específico para un cierto servicio, podemos enviar comandos a medida mediante el método `ExecuteCommand()` de `ServiceController`.

Éste toma como único parámetro un entero representando ese comando personalizado. El servicio, nuestra clase derivada de `ServiceBase`, invocará al método `OnCustomCommand()` cuando se reciba un comando de este tipo, dándonos opción a actuar en consecuencia.

Un controlador genérico

Básicamente ya conocemos toda la funcionalidad que puede ofrecernos la clase `ServiceController`, por lo que vamos a ver, en la práctica, cómo usarla. En principio diseñaremos un controlador genérico, similar a la utilidad de control de servicios del sistema aunque más sencillo. Así aprenderemos a recuperar los servicios existentes en el sistema.

Comenzaremos diseñando una interfaz de usuario como la de la figura 18.7. En el formulario se ha insertado un control `ListView`, que ocupa la mayor parte del espacio disponible, y un conjunto de botones en la parte inferior. El control `ListView` es como una lista o control `ListBox`, con la diferencia de que es capaz de mantener varios datos asociados por cada elemento. También puede mostrar iconos representativos de cada elemento, aunque ésta es una posibilidad que no vamos a usar.

Teniendo seleccionado el control `ListView` buscamos su propiedad `Columns` en la ventana **Propiedades** y pulsamos sobre el botón que aparece junto al valor, abriendo la ventana de definición de columnas. Pulsaremos tres veces el botón **Agregar** definiendo las columnas que aparecen en la figura 18.8, estableciendo su título y anchura.

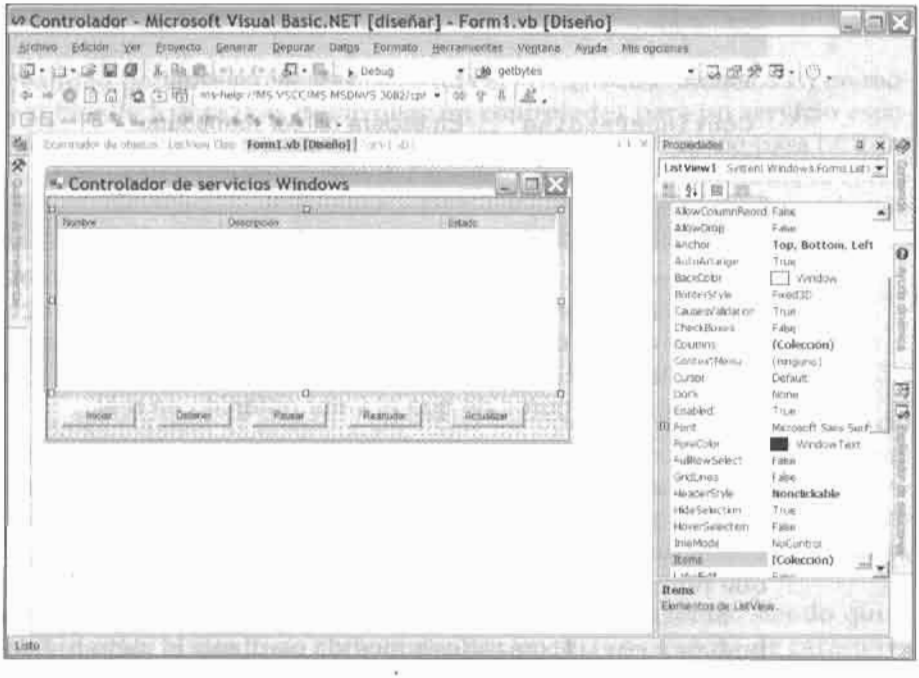


Figura 18.7. Aspecto del formulario al finalizar la fase de diseño

Figura 18.8. Definimos las tres columnas que aparecerán en la lista

Los botones estarán todos inicialmente desactivados, con la propiedad `Enabled` a `False`, a excepción del que hay más a la derecha, cuya función consiste en actualizar el estado de los servicios que aparece en la tercera columna del `ListView`. Tendremos que modificar la propiedad `Anchor` tanto de los botones, para que permanezcan en su posición relativa a la parte inferior del formulario, como del `ListView`, para que se ajuste en caso de que la ventana cambie de tamaño.

Finalizada la fase de diseño, tenemos que comenzar a introducir el código necesario asociado a los eventos del formulario, la lista y los botones. Lo primero que haremos será añadir la sentencia **Imports** `Syste.ServiceProcess` al inicio del módulo de código, para poder usar la clase `ServiceController` sin necesidad de estar componiendo referencias completas.

A continuación haremos doble clic sobre el formulario, o bien nos serviremos de las listas desplegables que hay en el editor de código, para abrir el método asociado al evento `Load`. Dicho método lo aprovecharemos para recuperar una lista de los servicios que hay en el sistema, añadiendo su nombre, descripción y estado al control `ListView`. El código necesario sería el siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
  Dim CServicio As ServiceController


  For Each CServicio In ServiceController.GetServices

    With ListView1.Items.Add(CServicio.ServiceName)
      .
      .SubItems.Add(CServicio.DisplayName)
      .SubItems.Add(CServicio.Status.ToString)
    End With
  Next
End Sub
```

Usamos el método compartido `GetServices()` de la clase `ServiceController` para recuperar un arreglo con todos los servicios del sistema que no están asociados con controladores de dispositivos, arreglo que recorremos para obtener cada uno de los servicios. Usamos las propiedades `ServiceName`, `DisplayName` y `Status` para obtener el nombre del servicio, su descripción y estado, respectivamente.

Nota

El control `ListView` dispone de una propiedad llamada `Items`, como los `ListBox`, a la que podemos añadir los elementos. Dichos elementos, sin embargo, no son simples cadenas de texto, sino componentes`ListViewItem`. Cada uno de ellos tiene una serie de propiedades, entre ellas una llamada `SubItems` que da acceso a los datos que aparecerán en las columnas de cada elemento.

Si ejecutáramos el programa en este momento ya veríamos aparecer la lista de servicios que hay en el sistema, pero no podríamos hacer nada más. El paso siguiente será controlar la selección de un servicio de la lista, recuperando su estado y actualizando consecuentemente los botones que hay en la parte inferior. El evento de `ListView` que nos interesa es `SelectedIndexChanged` y el código el mostrado a continuación:

```
' Cada vez que se cambie la selección en el ListView
Private Sub ListView1_SelectedIndexChanged( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ListView1.SelectedIndexChanged

  'Si no hay un elemento elegido, salimos
  If ListView1.FocusedItem Is Nothing Then
    Return  ' No hay nada
  End If

  'De ese elemento, tomamos el nombre del servicio
  Dim CServicio As New _
      ServiceController(ListView1.FocusedItem.Text)
  ' Y con ese nombre creamos un controlador de servicio
  btnIniciar.Enabled = _
    CServicio.Status <> ServiceControllerStatus.Running
  btnDetener.Enabled = CServicio.CanStop And _
    CServicio.Status = ServiceControllerStatus.Running
  btnPausar.Enabled = _
    CServicio.CanPauseAndContinue And _
    CServicio.Status = ServiceControllerStatus.Running
  btnReanudar.Enabled = _
  CServicio.Status = ServiceControllerStatus.Paused
End Sub
```

El control `ListView` dispone de una propiedad, llamada `FocusedItem`, que devuelve el objeto `ListViewItem` correspondiente al elemento que hay elegido en ese momento en la lista. Comprobamos que dicha referencia no sea `Nothing`, caso en el que no podríamos recuperar los datos necesarios y se produciría una excepción. Si `FocusedItem` apunta a un elemento utilizamos su propiedad `Text` para recuperar el nombre del servicio, nombre que usamos como parámetro para el constructor de `ServiceController` con el fin de obtener un controlador para el servicio. Acto seguido modificamos el valor de la propiedad `Enabled` de cada botón según el estado de dicho servicio.

Nota

Para poder importar el ámbito `System.ServiceProcess` desde este proyecto, que no es un servicio, debe añadir una referencia utilizando la opción adecuada del menú emergente asociado a la carpeta Referencias, en el Explorador de soluciones. Busque el ensamblado `System.ServiceProcess.dll` y añádalo.

En lugar de preparar un método independiente para el evento `Click` de cada uno de los botones, repitiendo parte del código ya que antes y después de iniciar o detener un servicio tendríamos que efectuar las mismas tareas, crearemos uno común para gestionar los cuatro botones de operación. En dicho método crearíamos un `ServiceController` asociado al servicio que se haya seleccionado en la lista, utilizándolo, en principio, para recuperar el estado actual del servicio, antes de operar sobre él. A continuación, y dependiendo del

botón que se haya pulsado, invocaremos al método que corresponda de `ServiceController`.

Antes de devolver el control nos serviremos de un bucle, como puede verse a continuación, para esperar hasta que el servicio haya cambiado de estado. Esperaremos aproximadamente un segundo como máximo, invocando a cada ciclo del bucle al método `Refresh()` del controlador para conocer el nuevo estado del servicio. Si éste no cambiara significará que el servicio no responde, lo cual notificamos adecuadamente.

```
' Este método destroza los controles nativos y de .Net
Private Sub btnIniciar_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnIniciar.Click, _
   btnDetener.Click, btnPausar.Click, btnReanudar.Click

  ' Recuperamos el servicio sobre el cual actuar
  Dim CServicio As New _
      ServiceController(ListView1.FocusedItem.Text)
  ' y guardamos su estado actual
  Dim Estado As ServiceControllerStatus = CServicio.Status
  Dim Espera As Integer

    Según el botón que se haya pulsado
  Select Case CType(sender, Button).Name
    Case "btnIniciar"
      CServicio.Start() ' llamamos a un método
    Case "btnDetener"
      CServicio.Stop() ' a otro del controlador
    Case "btnPausar"
      CServicio.Pause()
    Case "btnReanudar"
      CServicio.Continue()
  End Select

  ' esperamos un máximo de un segundo hasta que el
  ' estado del servicio cambie
  Do While Estado = CServicio.Status
    Espera += 1
    If Espera = 10 Then ' si no llega a cambiar
      ' mostramos un aviso
      MessageBox.Show("No es posible actuar sobre este servicio")
      Exit Do
    End If
    ' esperamos una décima de segundo en cada ciclo
    System.Threading.Thread.CurrentThread.Sleep(100)
    ' y actualizamos el estado del servicio
    CServicio.Refresh()
  Loop

  ' al finalizar mostramos en la lista el nuevo estado
  EstadoServicios() ' de los servicios
End Sub
```

Como puede ver en este código, lo último que hace el método de gestión de los botones es llamar a un método llamado `EstadoServicios()`. Al pulsar el botón **Actualizar** también llamaremos a dicho método, que será el encargado

de recorrer la lista de servicios y actualizar el estado leyéndolo de nuevo. El código de ese método es éste:

```
Private Sub EstadoServicios()
  Dim Elemento As ListViewItem
  Dim CServicio As ServiceController

  For Each Elemento In ListView1.Items
    CServicio = New ServiceController(Elemento.Text)
    Elemento.SubItems(2).Text = CServicio.Status.ToString
  Next

  ListView1_SelectedIndexChanged(Me, EventArgs.Empty)
End Sub
```

Con esto ya tenemos finalizado nuestro controlador de servicios. Al ejecutarlo verá aparecer la lista de servicios del sistema. Seleccionando cualquiera de ellos los botones de operación se activarán o desactivarán según las acciones posibles. Por último, al pulsar cualquiera de dichos botones se enviará al servicio el comando correspondiente.

En la figura 18.9 puede ver la aplicación en funcionamiento, justo después de haber puesto en marcha el servicio `MSSQLSERVER`.

Nota

Tenga cuidado al utilizar esta aplicación, o la utilidad de gestión de servicios del sistema, y no manipule servicios que desconoce, ya que puede afectar al funcionamiento de su sistema.

Un controlador para `ServicioHora`

La aplicación que acabamos de desarrollar es de uso genérico, en el sentido de que no opera sobre un servicio concreto. Dado que el propio sistema operativo ya cuenta con utilidades y comandos que permiten efectuar esas operaciones sobre cualquier servicio, no tiene mucho sentido que nosotros desarrollemos algo parecido aunque, como hemos visto en el punto anterior, podemos hacerlo perfectamente.

Utilizando exactamente los mismos recursos que hemos conocido, la clase `ServiceController` y sus miembros, podríamos crear un controlador específico para nuestro servicio. Al ser ejecutado, este controlador mostraría en el extremo derecho o inferior de la **Barra de tareas** de Windows un icono de notificación. Al desplazar el cursor del ratón sobre él indicaría, mediante un men-

saje de texto, el estado actual del servicio (véase figura 18.10), mientras que al hacer clic invertiríamos su estado, es decir, si está detenido lo pondríamos en marcha y viceversa. Por último, haciendo doble clic sobre el icono cerraríamos el controlador.

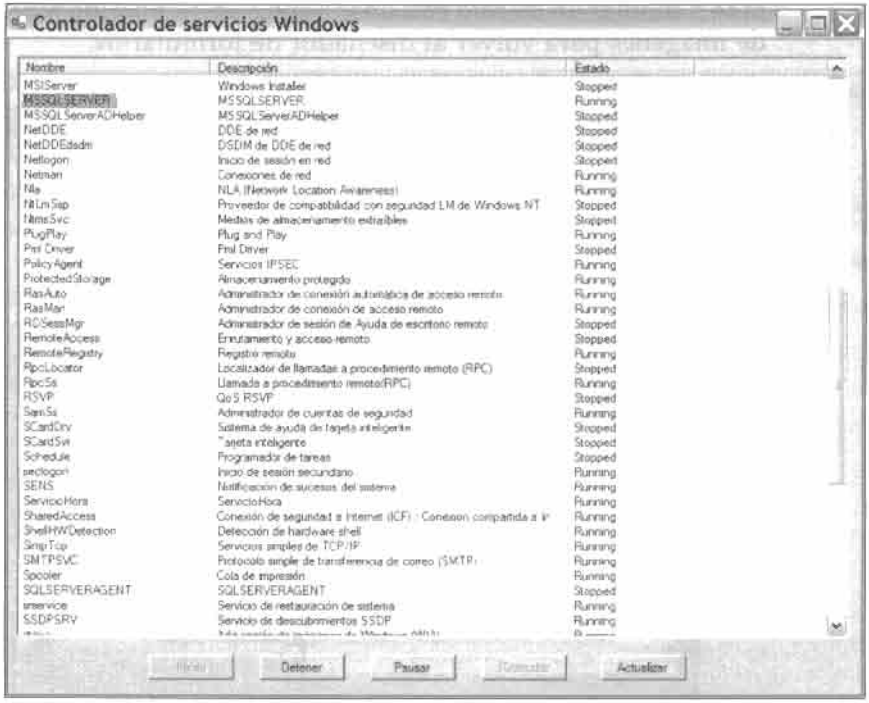


Figura 18.9. Aspecto del controlador de servicios en funcionamiento

Figura 18.10. El controlador específico muestra un icono en la Barra de tareas

Para desarrollar este controlador partiremos de una nueva aplicación basada en formularios Windows. Lo primero que haremos será añadir al proyecto un Archivo de icono, como se muestra en la figura 18.11. El archivo se abrirá en un diseñador específico para iconos, el que aparece en la figura 18.12, donde daremos rienda suelta a nuestra creatividad. En nuestro caso nos hemos limitado a dibujar un círculo con unas agujas en su interior. Puede cerrar el editor de imágenes para volver al diseñador de formularios.

Nota

Existe una barra de botones específica para el diseño de imágenes que, inicialmente, no está visible. Use el menú emergente de cualquiera de las barras visibles para acceder a ella.

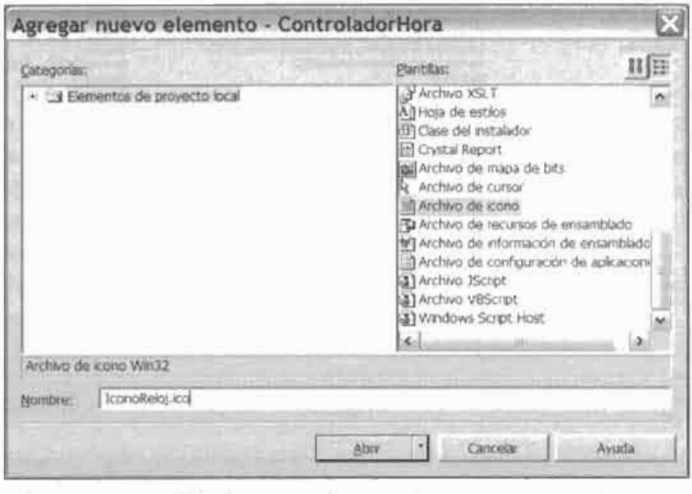


Figura 18.11. Añadimos un icono al proyecto actual

A continuación tomaremos del Cuadro de herramientas un control `NotifyIcon` y lo añadiremos a nuestro formulario. En la ventana Propiedades verá que este componente cuenta con una serie de propiedades que permiten establecer si será o no visible, el texto asociado, icono y menú emergente. Use la propiedad `Icon` para recuperar el icono que acaba de diseñar, facilitando el camino completo al archivo. Aquí termina el trabajo de diseño propiamente dicho, el resto del trabajo se efectuará mediante código.

En principio no nos interesa que al ejecutar esta aplicación aparezca el formulario, dado que la interfaz de usuario será el icono de notificación, sin más. Utilizando la ventana de propiedades del proyecto, que ya conocemos, estableceremos que el punto de entrada a la aplicación no será el formulario sino el método `Main()`, un método que deberemos añadir a la clase `Form1`. También daremos el valor `False` a la propiedad `ShowInTaskBar` del formulario, para evitar la aparición de un botón de acceso al programa en la Barra de tareas.

```
Public Shared Sub Main()

  Dim MiFormulario As New Form1()

  Application.Run()
End Sub
```

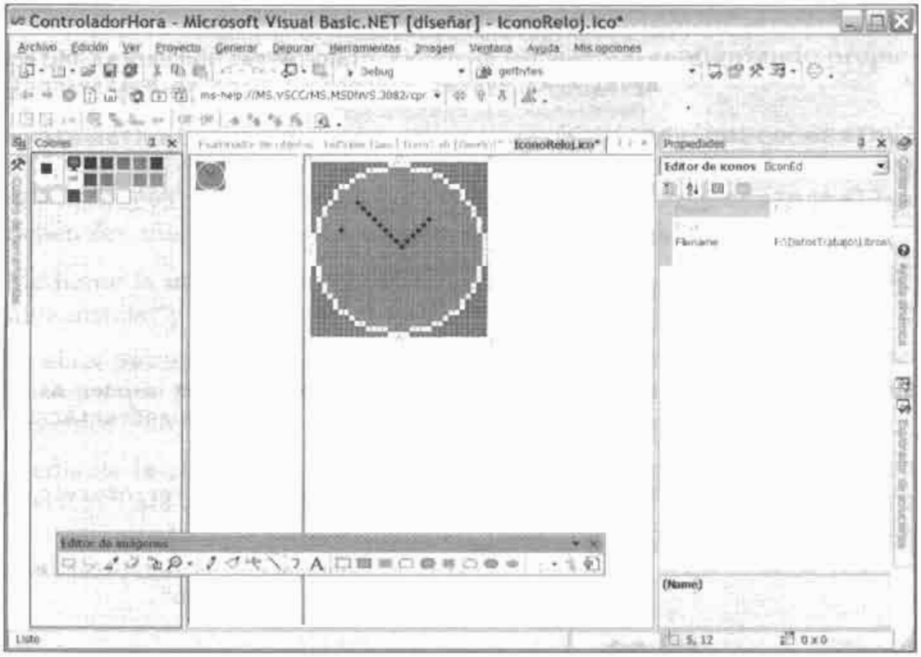


Figura 18.12. Visual Basic .NET dispone de un diseñador de imágenes

Figura 18.13. Modificamos las opciones del proyecto para iniciar la ejecución en `Main()`

Creamos el formulario, poniendo así en marcha la configuración del componente `NotifyIcon`, y después ponemos en marcha la ejecución del programa pero sin mostrar el formulario. En ese momento el icono ya estará visible en la Barra de tareas, puesto que dimos el valor `True` a su propiedad `Visible`. Lo único que tenemos que hacer ahora es responder a tres de los eventos del control: `Click`, `MouseMove` y `DoubleClick`. El código de los métodos asociados a estos eventos sería el siguiente:

```
Private Sub NotifyIcon1_Click(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles NotifyIcon1.Click

  Dim MiServicio As New ServiceController("ServicioHora")

  If MiServicio.Status = ServiceControllerStatus.Running Then
    MiServicio.Stop()
  Else
    MiServicio.Start()
  End If
End Sub


Private Sub NotifyIcon1_MouseMove(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.MouseEventArgs) _
    Handles NotifyIcon1.MouseMove

  Dim MiServicio As New ServiceController("ServicioHora")

  NotifyIcon1.Text = IIf( _
    MiServicio.Status = ServiceControllerStatus.Running, _
    "Servicio horario en funcionamiento", _
    "Servicio horario detenido")
End Sub


Private Sub NotifyIcon1_DoubleClick(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles NotifyIcon1.DoubleClick
  Application.Exit()
End Sub
```

No es necesario hacer nada más. Si ejecuta el proyecto comprobará que su funcionamiento se ajusta al objetivo que exponíamos al principio de este punto. Tenemos un controlador específico para nuestro servicio, de tal forma que podemos conocer y modificar su estado sin necesidad de acceder a la utilidad de gestión de servicios del sistema ni la consola o línea de comandos.

Puntos clave

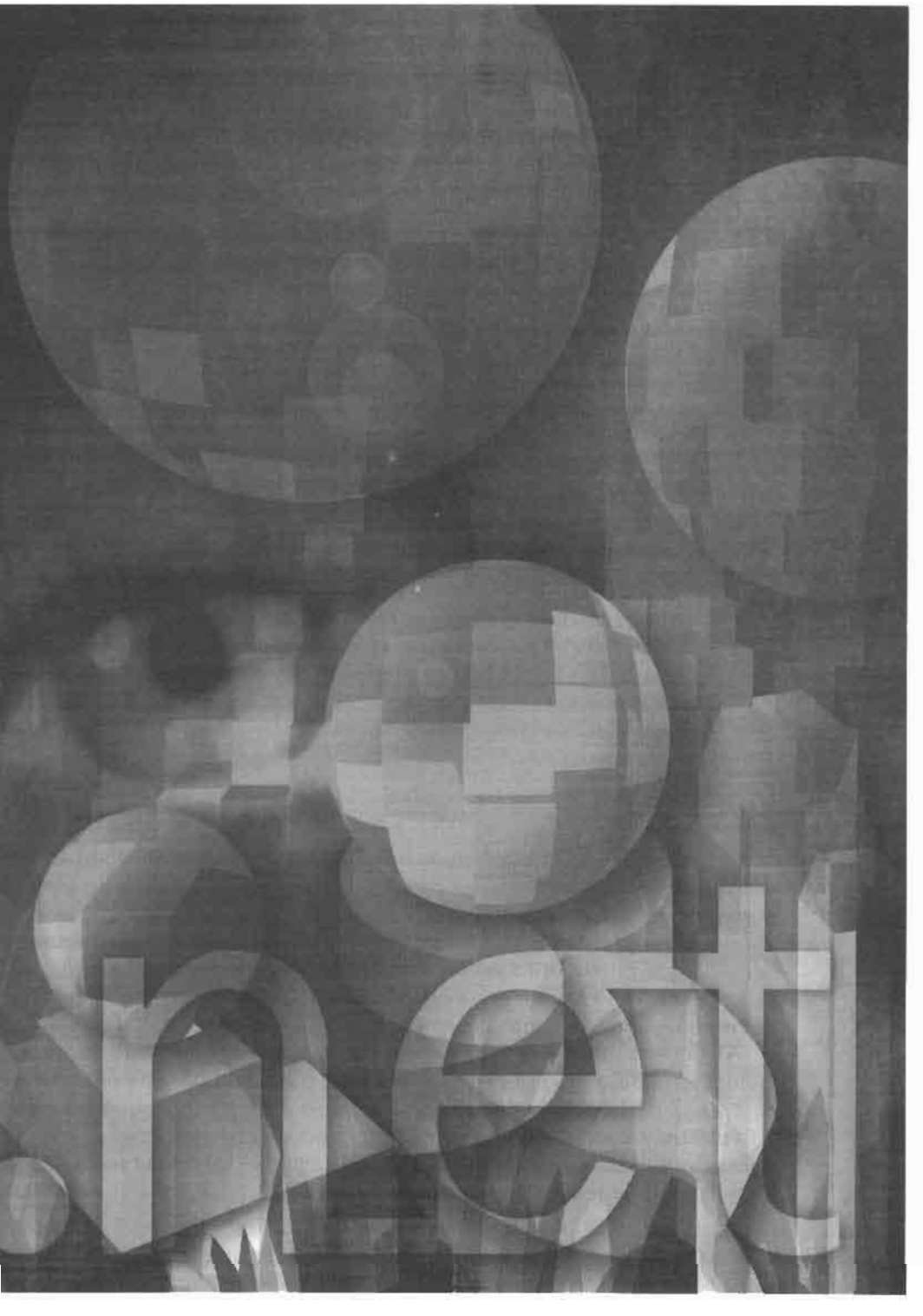
- Para crear un servicio es necesario definir una nueva clase derivándola de `ServiceBase`, de manera análoga a como definimos un nuevo formulario partiendo de la clase `Form`.

- Un mismo proyecto puede constar de varias derivadas de `ServiceBase`, apareciendo cada una de ellas como un servicio distinto aunque alojados de manera compartida en un mismo módulo.
- En el mismo módulo en el que están los servicios es necesario incluir una clase que actúe como instaladora. Dicha clase se derivará de `System.Configuration.Install.Installer` y deberá tener el atributo `RunInstaller` con el valor `True`.
- La clase que actúa como instaladora deberá crear un objeto `ServiceInstaller` por cada servicio que exista en el módulo, configurando propiedades como el nombre o el modo de activación.
- En esa misma clase también debe crearse un objeto `ServiceProcessInstaller` para configurar el módulo que contiene a los servicios. Tanto los objetos `ServiceInstaller` como el objeto `ServiceProcessInstaller` deben ser añadidos a una colección llamada `Installers`.
- Mediante la utilidad `installutil`, desde la línea de comandos, podemos instalar y desinstalar los servicios que hayamos desarrollado.
- La clase `ServiceBase` dispone de propiedades que determinan las operaciones que pueden efectuarse sobre el servicio, así como métodos que debemos redefinir para responder adecuadamente a esas operaciones.
- Mediante la clase `ServiceController` es posible controlar cualquier servicio. Para ello dispone de métodos como `Start()`, `Stop()`, `Pause()` y `Continue()`, que inician, detienen, pausan o reanudan el servicio asociado al objeto.

Resumen

Tras leer este capítulo hemos aprendido prácticamente todo lo relativo al diseño de servicios Windows con Visual Basic .NET, habiendo visto también cómo los asistentes con que cuenta generan gran parte del código necesario. Hemos desarrollado un servicio simple y comprobado su funcionamiento, un servicio que puede utilizar como base para crear otros a su medida. Finalmente, se ha descrito la clase `ServiceController` y mostrado cómo usarla para crear controladores tanto genéricos como específicos.

Al tiempo que hemos cubierto la materia propia de este capítulo, en sus ejemplos también hemos aprendido a usar, aunque básicamente, los servicios de *sockets* de la plataforma .NET, así como componentes que no conocíamos: `ListView` y `NotifyIcon`. Será así en la mayoría de los casos, en la práctica al necesitar una cierta funcionalidad, como vaya conociendo el resto de los controles y componentes de que dispone Visual Basic .NET.



19 Desarrollo de servicios Web

Los clientes actuales que existen en la Web, ya sean usuarios físicos o aplicaciones de terceros, son consumidores de servicios. Conseguir que la actual red, compuesta principalmente de páginas HTML y archivos, se convierta en una red de servicios es uno de los objetivos de la plataforma .NET. Un servicio Web es, como su propio nombre indica, un servicio que una determinada empresa pone a disposición de los usuarios. Dependiendo de las necesidades, puede tratarse de un servicio de uso interno o accesible para el público de Internet en general. Para crear un servicio Web podemos usar cualquiera de los lenguajes disponibles en la plataforma .NET, mientras que los consumidores pueden potencialmente estar en cualquier lugar y ejecutándose sobre cualquier hardware y sistema operativo. Realmente, cualquiera puede ofrecer o consumir un servicio Web si utiliza el protocolo adecuado, no es necesario disponer de Windows, la plataforma .NET o el lenguaje Visual Basic .NET.

Aunque existen implementaciones que permiten publicar servicios Web sobre otros sistemas, gracias a IBM es posible, por ejemplo, usar Apache, en este capítulo nos centraremos en el uso de IIS sobre Windows. Los consumidores pueden usar una simple solicitud HTTP, desde un cliente Web, o bien usar un protocolo RPC sobre XML conocido como SOAP (*Simple Object Access Protocol*).

¿Qué es un servicio Web?

Actualmente raro es el usuario habitual de informática que no sabe lo que es una página Web. Si le hablamos de un servicio Web, sin embargo, seguro

que la mayoría de ellos no lo tendrán muy claro. Es lógico al tratarse, como ya se ha dicho, de un concepto relativamente nuevo. ¿Qué es en realidad un servicio web? ¿Cuál es su aspecto? ¿Cómo podemos crear uno o utilizarlo? Estas son muchas de las cuestiones que surgen en un principio.

Imagine que usted adquiere un ordenador nuevo y que en él tan sólo encuentra inicialmente un sistema operativo, nada más. Para efectuar su trabajo, sin embargo, seguramente necesitará una libreta de direcciones, una calculadora y una aplicación de cálculo de estructuras, por poner algunos ejemplos. Obviamente, pensará, lo que tiene que hacer es buscar y adquirir el software que necesita, instalándolo en su ordenador para usarlo cuantas veces necesite.

Este es el modo en que *funcionamos* actualmente con nuestros ordenadores. Periódicamente, además, solemos actualizar todo ese software que hemos adquirido para disponer de una serie de posibilidades que, en ocasiones, ni siquiera necesitamos.

Servicios y aplicaciones

En cierta forma, una aplicación es un servicio o un conjunto de servicios instalados en nuestro sistema. Microsoft Excel, por hablar de un programa bien conocido, es una aplicación formada por varios componentes o servicios distintos que facilitan la edición de los libros, diseño de gráficos, etc.

Suponga que los distintos componentes que forman Microsoft Excel no están instalados en su ordenador sino en un servidor Web, por ejemplo en la sede de Microsoft, y que para usarlos lo que hace es enviar solicitudes mediante su navegador habitual. Los componentes serían servicios Web, la sede de Microsoft sería la que expondría dichos servicios y, finalmente, nosotros actuaríamos como consumidores.

La aparición de los servicios Web va a influir en la forma de hacer negocio de muchas empresas. Con la expansión de las redes empresariales y su interconexión a través de Internet, serán muchos los que opten por alquilar u ofrecer sus productos en forma de servicios Web, en vez de hacerlo como hasta ahora, vendiendo licencias de uso de esos productos. A las empresas consumidoras, asimismo, puede interesarles más pagar por el uso de esos servicios que adquirir las aplicaciones equivalentes, sabiendo que siempre van a estar usando las últimas versiones sin necesidad de actualizar sus programas.

Para que este mecanismo de oferta de servicios tenga éxito, no obstante, antes es necesario que las distintas empresas implicadas lleguen a acuerdos de estándares para la descripción de servicios, la comunicación con ellos, la búsqueda, etc.

Universalidad de un servicio

Actualmente, al publicar un documento en un servidor Web no nos preocupamos demasiado sobre la plataforma hardware, el sistema operativo ni las aplicaciones que utilizará el cliente para acceder a él. Esto es posible porque

HTML es un estándar universal, lo mismo que el protocolo HTTP utilizado para solicitar y transmitir esos documentos. De esta forma se consigue que un cierto servicio, en este caso la Web, sea realmente universal, ya que está accesible para todos sin limitaciones.

Conseguir que los servicios Web alcancen ese mismo nivel de universalidad es uno de los objetivos de empresas como Microsoft e IBM que, conjuntamente con muchas otras, están definiendo protocolos y borradores de estándares que permitan describir un servicio, publicarlo para que los consumidores puedan encontrarlo, establecer una comunicación con él por parte de los clientes, etc.

Aunque sea algo reiterativo, hay que incidir en el hecho de que es posible crear un servicio Web utilizando cualquier lenguaje, sobre cualquier sistema operativo y arquitectura o plataforma hardware. Para conseguir que ese servicio sea accesible para los consumidores hay que describirlo usando un lenguaje estándar que es WSDL (*Web Service Description Language*). El cliente podrá estar creado también en cualquier lenguaje y ejecutarse sobre cualquier sistema operativo y hardware, obteniendo la descripción WSDL de un servicio de directorio conocido como UDDI (*Universal Description, Discovery and Integration*). Con esta información se comunicaría con el servicio usando protocolos como HTTP o SOAP.

Protocolos y lenguajes

Los servicios Web tienen como objetivo hacer más realidad que nunca la meta perseguida por la computación distribuida, para la cual se han utilizado hasta ahora tecnologías como CORBA, DCOM o Java RMI. El problema es poder efectuar llamadas a procedimientos remotos con un protocolo RPC (*Remote Procedure Call*) que no dependa de un cierto sistema, como es el caso de DCOM, o lenguaje, como ocurre con RMI.

Hace un par de años tres empresas, Developmentor, UserLand y Microsoft, propusieron un protocolo que permitía efectuar llamadas RPC usando el lenguaje XML para describir la llamada y parámetros y el protocolo HTTP como transporte. El uso de HTTP para el transporte de las llamadas salva muchos obstáculos con los que se encontraban CORBA y DCOM, ya que la mayoría de las empresas utilizan cortafuegos que impiden la entrada en sus servidores por puertos no estándar.

El puerto usado para HTTP, la base de la Web, es uno de los considerados estándar y, por tanto, ideal para ser utilizado por SOAP.

Dado que SOAP describe sus llamadas y respuestas utilizando XML, en la práctica es posible utilizarlo desde cualquier sistema operativo y lenguaje. Actualmente existen diversas implementaciones que permiten utilizar SOAP desde Windows, Unix y Linux.

Utilizando SOAP cualquier cliente puede consumir un servicio ofrecido por un proveedor, como puede verse en la figura 19.1. Es necesario, no obstante, que el cliente conozca de antemano la localización y naturaleza del servicio, sabiendo a qué métodos puede llamar, con qué parámetros y qué tipo de respuesta obtendrá.

Figura 19.1. Un consumidor usa SOAP para, a través de Internet, acceder a un servicio ofrecido por un proveedor

La descripción de un servicio se efectúa, como se dijo antes, en WSDL (*Web Service Description Language*). Un módulo WSDL es un archivo XML en el que se identifica el servicio y se facilita el esquema para poder utilizarlo, generalmente entregándose información sobre los distintos protocolos que es posible utilizar. Más adelante veremos cómo obtener el WSDL correspondiente a un servicio Web creado por nosotros, ya que ASP.NET es capaz de generar esa descripción automáticamente.

Resueltos los problemas de conectividad entre consumidor y proveedor y descripción de los servicios, nos encontramos con un último obstáculo: cómo encontrará el consumidor a los proveedores que ofrecen los servicios que él necesita. Traslade la situación a otro entorno. Piense que necesita los servicios de un fontanero, ¿dónde buscaría lo que necesita? Efectivamente, en algún tipo de guía o páginas amarillas.

Tras llegar al acuerdo de creación del lenguaje WSDL, las tres empresas citadas antes anunciaron a principios de septiembre de 2000 que estaban trabajando en un servicio llamado UDDI. Se trata de un servicio de directorio a nivel mundial en el que los proveedores de servicios, de manera gratuita, podrán registrar sus productos. Los consumidores usarán UDDI para consultar ese registro a la búsqueda de lo que necesitan. En él encontrarán una descripción de los servicios, información sobre el proveedor y todo lo necesario para contactar con él.

Mientras que algunos protocolos y lenguajes están ya finalizados y propuestos como estándar, es el caso de SOAP, otros por el contrario están en plena fase de desarrollo y evolución, como WSDL y UDDI. La parte más importante de este capítulo está centrada en la creación y consumo de servicios usando Visual Basic .NET.

En los puntos siguientes encontrará una introducción a los protocolos y lenguajes más importantes relacionados con los servicios Web. Si desea más información sobre ellos visite las sedes de Microsoft, IBM y el W3C.

Introducción a XML

Como antes ocurriera con otras tecnologías surgidas de Internet, los desarrolladores están encontrando en XML múltiples aplicaciones. Si aún cree que XML es una variante de HTML o que tan sólo sirve para componer documentos para la Web, le interesa seguir leyendo.

No es que XML (*eXtensible Markup Language*, Lenguaje de marcas extensible) sea algo realmente nuevo, pero aún así sigue existiendo una cierta confusión en torno a él. Son muchos los que piensan que XML es una variante o un derivado de HTML, y otros tantos creen que tan sólo es útil para insertar tablas de datos en las páginas HTML. No obstante, XML puede marcar, y de hecho está haciéndolo, el futuro en campos tan importantes como las bases de datos o el comercio electrónico, prueba de ello es la apuesta en este estándar por parte de centenares de empresas de todos los sectores.

Como programador, posiblemente tenga que vérselas con XML en un plazo relativamente corto, si es que no ha tenido que hacerlo ya. Mediante XML el intercambio de información, entre distintas aplicaciones, sistemas y plataformas, es mucho más fácil. El diseño de servicios Internet para acceso a bases de datos empresariales, una necesidad cada vez más presente, también es una tarea más sencilla usando XML, conjuntamente con hojas de estilo XSL (*eXtensible Stylesheet Language*).

Tanto XML como XSL son estándares del W3C, el consorcio que rige muchos de los estándares que se usan en Internet y son aceptados internacionalmente. Si desea mantenerse al día respecto a XML, y otros estándares relacionados, anote en su lista de páginas web la dirección `http://www.w3.org`.

En los puntos siguientes encontrará la información necesaria para saber qué es XML, qué es un *parser* XML, cómo crear documentos XML bien formados y cómo comprobar su validez. También conoceremos lo que son los documentos de definición y las hojas de estilo, con las que podremos generar HTML a partir de documentos XML.

¿Qué es XML?

Como su propio nombre indica, XML es un lenguaje de etiquetas extensible, es decir, en el que podemos crear nuestras propias marcas o etiquetas. Las etiquetas son la base de XML, al igual que ocurre con HTML. No es de extrañar, ya que ambos son lenguajes derivados de SGML (*Standard Generalized Markup Language*). El uso de etiquetas, no obstante, es uno de los pocos puntos comunes que hay entre XML y HTML.

La finalidad de un documento XML es definir la estructura de una información, contenida asimismo en el propio documento. Las etiquetas XML no indican, por el contrario, cuál es el formato visual de la información.

HTML es un lenguaje de marcas, pero su finalidad es tan sólo establecer la apariencia de unos datos contenidos en el documento, en ningún caso describir su estructura.

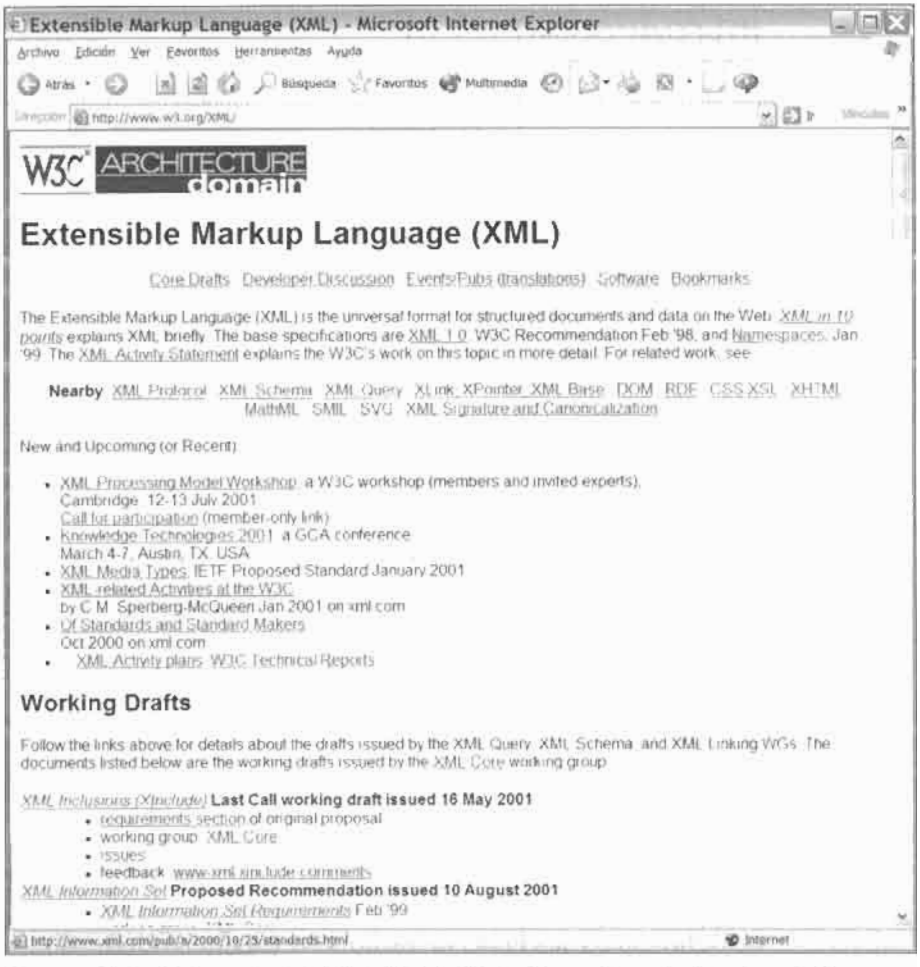


Figura 19.2. Sede web del *World Wide Web Consortium* dedicada a XML

Usando XML, el usuario puede definir sus propias etiquetas según la información cuya estructura necesite describir. Esto significa que no existe un límite preestablecido, en la propia definición del lenguaje, que actúe como frontera en cuanto a los datos que es posible describir. HTML, por el contrario, cuenta con un conjunto bien conocido de etiquetas, con una finalidad preestablecida y no ampliable por el usuario final.

Trabajo con etiquetas

Tanto un documento XML, como uno HTML, contienen datos y etiquetas. Las etiquetas HTML afectan a la apariencia de los datos, presumiblemente de una forma independiente del dispositivo final en que se muestren. Las etiquetas XML no indican cómo debe mostrarse un dato, sino qué es el dato.

Observe el listado siguiente, correspondiente a un documento HTML, y el que hay inmediatamente detrás, que tiene la misma información pero en XML.

A no ser que se tengan conocimientos de HTML, será difícil saber cuál es el significado de etiquetas como <p> o <li>. En el documento XML, por el contrario, las etiquetas son autodescriptivas y, en consecuencia, es fácil apreciar que la información es una lista de productos clasificados en varios grupos.

```
<html>
<body>
LISTA DE PRODUCTOS
<p>Software
<ul>
<li>
Visual Basic .NET</li>
<li>
Visual C++ .NET</li>
<li>
VisualAge C# .NET</li>
<li>
Visual J# .NET</li>
</ul>

<p>Libros
<ul>
<li>
Programación en Windows 2000</li>
<li>
Cómo programar con Visual Basic para torpes</li>
<li>
Programación con Visual Basic .NET</li>
<li>
Programación con Visual Studio .NET</li>
</ul>
</body>
</html>
```

Como puede ver a continuación, el documento XML es autodescriptivo:

```
<?xml version="1.0"?>
<ListaProductos>
  <Software>
    <Producto>Visual Basic .NET</Producto>
    <Producto>Visual C++ .NET</Producto>
    <Producto>VisualAge C# .NET</Producto>
    <Producto>Visual J# .NET</Producto>
  </Software>
  <Libros>
    <Libro>Programaci&#243;n en Windows 2000</Libro>
    <Libro>C&#243;mo programar con Visual Basic para torpes</Libro>
    <Libro>Programaci&#243;n con Visual Basic .NET</Libro>
    <Libro>Programaci&#243;n con Visual Studio .NET</Libro>
  </Libros>
</ListaProductos>
```

Visualización de los documentos

Para crear un documento tan simple como el propuesto y ver su estructura, no es necesario utilizar ninguna herramienta específica. El Bloc de notas de

Windows es una utilidad suficiente para este trabajo. No obstante, siempre puede utilizar el propio editor de Visual Basic .NET ya que, como se aprecia en la figura 19.3, reconoce la sintaxis de HTML y XML, destacando sintácticamente los diferentes elementos.

Cuando se trabaja con documentos HTML y XML, el editor de Visual Basic .NET muestra normalmente unas pestañas, en la parte inferior, que nos permite alternar entre el código propiamente dicho y una previsualización del documento resultante, en el caso de HTML, o bien la tabla de datos que contiene el documento XML. De hecho, puede usar la vista previa de HTML para crear el documento visualmente, sin necesidad de introducir marcas y, de igual forma, la pestaña **Data** del editor XML hace posible la introducción de nuevos datos en el documento de manera mucho más simple.

Figura 19.3. Edición de documentos HTML y XML en el editor de Visual Basic .NET

En la figura 19.4 puede ver el documento HTML, correspondiente al primer listado, abierto en la página **Diseño** del editor de Visual Basic .NET. El aspecto es el de una lista de productos agrupados en categorías, tal y como se esperaba. El aspecto que tendría este documento en otro cliente, por ejemplo Netscape Navigator, sería prácticamente idéntico, ya que las etiquetas HTML especifican cuál debe ser la apariencia de la información.

En la parte inferior de la misma figura aparece el documento XML en la página **Data** del editor XML de Visual Basic .NET. El resultado, como puede verse, es bien distinto. La plataforma .NET incorpora un analizador XML y otras herramientas que Visual Basic .NET utiliza para comprobar la estructura del documento y, en caso de que esté *bien formado*, recuperar su contenido y facilitar la edición. La lista de datos, además, es dinámica. Puede abrir y cerrar las distintas ramas que existen.

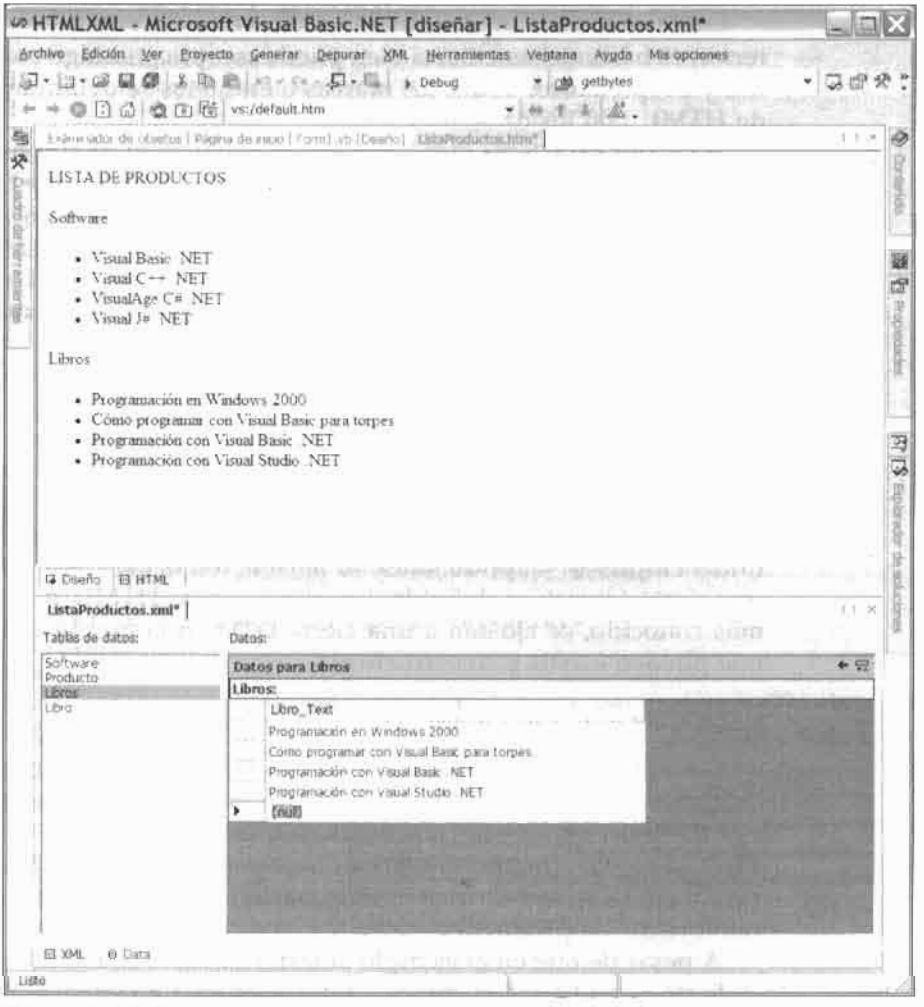


Figura 19.4. Edición visual de los documentos HTML y XML

Gracias a la capacidad de los editores de Visual Basic, no será necesario recurrir a herramientas externas para trabajar con documentos HTML o XML.

Nota

Al introducir código XML en el editor de Visual Basic .NET, tras crear el documento con el correspondiente elemento de la ventana Agregar nuevo elemento, verá que al abrir una marca, por ejemplo `<Producto>`, automáticamente aparece el cierre y el cursor queda entre ambas etiquetas, esperando la introducción de información.

Estructura de un documento XML

La estructura de un documento XML, como ha podido ver en el ejemplo anterior, es bastante sencilla, si bien puede complicarse algo con el uso de entidades y propiedades. Todas las marcas o etiquetas se delimitan, como en el caso de HTML, con los caracteres < y >.

Si los dos caracteres mencionados se acompañan de signos de cierre de interrogación, como en el caso de la etiqueta `<?xml version="1.0"?>`, la marca adquiere un significado especial. A este tipo de marcas se las denomina de proceso, sirviendo para indicar, por ejemplo, la versión de XML a la que se ajusta el documento, el conjunto de caracteres utilizado, etc. La versión actual, y única existente, de XML es la 1.0, de ahí que por el momento sea esa la única versión que pueda utilizarse.

Tras la etiqueta que identifica al documento como un documento XML, que es la primera, el resto de la información se compondrá de etiquetas y datos. Las etiquetas pueden ser creadas por nosotros, como en el caso del ejemplo anterior, o bien tomarse de una DTD (*Document Type Definition*, Definición de tipo de documento) o un esquema XSD ya existente. La existencia de DTDs y esquemas públicos y estándar facilitan el intercambio de información entre distintas empresas y aplicaciones, al ajustar todas ellas sus documentos XML a una cierta estructura definida. Los documentos HTML, por mencionar el caso más conocido, se ajustan a una cierta DTD en la que se indican las etiquetas que pueden existir y su estructura.

Estructura de los datos

Los datos de un documento XML se estructuran siempre de forma jerárquica. Esto significa que existe un elemento raíz, en el caso de nuestro documento es `<ListaProductos>`, que contiene a los demás: `<Software>` y `<Libros>`. Éstos, a su vez, pueden contener otros elementos. Entre las etiquetas de apertura y cierre se introducirá la información que, en el ejemplo propuesto, es el nombre de los productos.

A pesar de que en el ejemplo anterior tan sólo existen dos niveles de datos, lo cierto es que pueden crearse tantos como se necesiten. Observe el documento XML siguiente, que es una ampliación del anterior. En él se detallan ahora

ciertos datos de cada producto, como el nombre, descripción, fabricante, autor o editorial, según los casos. Fíjese en el sangrado del texto que, a pesar de no ser algo obligatorio, aclara bastante la estructura durante la edición. Gracias a este sangrado es fácil ver que la etiqueta <Distribuidores> cuenta con una lista de datos, por poner un ejemplo. No obstante, la manera más clara de ver la estructura del documento consiste en abrirlo en Internet Explorer, como se ha hecho en la figura 19.5. Para ello no tiene más que usar la opción Ver>Abrir con ... y seleccionar el elemento correspondiente a Internet Explorer.

```
<?xml version="1.0" encoding="utf-8" ?>
<ListaProductos>
  <Software>
    <Producto>
      <Nombre>Visual Basic .NET</Nombre>
      <Descripcion>Entorno de desarrollo RAD basado
            en BASIC</Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>Danysoft</Distribuidor>
        <Distribuidor>Database DM</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual C++ .NET</Nombre>
      <Descripcion>Compilador de C++ para la plataforma
            .NET</Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>DanySoft</Distribuidor>
        <Distribuidor>Action</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>Visual C# .NET</Producto>
    <Producto>Visual J# .NET</Producto>
  </Software>
  <Libros>
    <Libro>
      <Titulo>Programación en Windows 2000</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>
      <Titulo>Como programar con Visual Basic para torpes</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>Programación con Visual Basic .NET</Libro>
    <Libro>Programación con Visual Studio .NET</Libro>
  </Libros>
</ListaProductos>
```

Conjuntos de caracteres en XML

Los documentos XML pueden encontrarse codificados en distintos conjuntos de caracteres, siendo el asumido por defecto UTF-8, que es un Unicode de

8 bits. En caso de utilizarse cualquier otra codificación, será preciso indicarlo en la etiqueta de proceso que hay al inicio del documento.

Figura 19.5. Aspecto del documento XML abierto en Internet Explorer

Si observa los documentos XML correspondientes a los ejemplos anteriores, podrá ver que en el primero no existen caracteres acentuados. Éstos han sido sustituidos por *entidades* como `ó` que representan al carácter que desea incluirse. Alternativamente, podría haberse utilizado `ó`.

Lo anterior es necesario porque en la codificación por defecto no tienen cabida algunos caracteres europeos, como las vocales con acentos graves o agudos o la Ñ. Si deseásemos incluir estos caracteres directamente, tendríamos que especificar de manera explícita una codificación que los permitiese.

Aunque utilizar una codificación diferente es algo perfectamente válido, en ocasiones nos podemos encontrar con problemas si un cierto analizador no reconoce ese conjunto de caracteres. La solución pasa por usar siempre `UTF-8`, sustituyendo los mencionados caracteres por sus códigos.

Documentos XML bien formados

A diferencia de lo que ocurre con los documentos HTML, en los que existe una cierta permisividad en cuanto a su estructura, los documentos XML deben estar *bien formados* para que su proceso sea posible. Para conseguir que un documento XML esté bien formado tan sólo hay que seguir unas reglas básicas, reglas que, además, podrían aplicarse también a HTML.

Un documento XML debe iniciarse siempre con la etiqueta de proceso que le identifica como XML y en la que, además, se indica la versión, el conjunto de caracteres utilizado, etc. A continuación tiene que existir una etiqueta que actúe como raíz, en la que estarán contenidas todas las demás. En cada documento XML tan sólo puede existir una raíz, en caso contrario no estará bien formado.

Todas las etiquetas XML deben tener su pareja, es decir, toda apertura debe contar con un cierre. Hay que tener en cuenta, además, que no pueden existir solapamientos entre las aperturas y cierres, sino que cada pareja de marcas debe contener totalmente a los datos que define. Por ello, los dos fragmentos de código XML no estarían bien formados.

```
<?xml version="1.0"?>
<ListaProductos>
  <Software>
    <Producto>Visual Basic .NET
  </Software>
</ListaProductos>
```

```
<?xml version="1.0"?>
<ListaProductos>
  <Software>
    <Producto>Visual Basic .NET
  </Software></Producto>
</ListaProductos>
```

En el primer caso la etiqueta `<Producto>` no cuenta con la correspondiente `</Producto>`, mientras que en el segundo el orden de las etiquetas no es el correcto, ya que `<Software>` debería contener totalmente al grupo de datos `<Producto>`. Al intentar procesar este código, simplemente cambiando a la página **Data** del editor, el analizador XML generará un error. En la figura 19.6 puede ver cómo se indica que el documento no está bien formado, comunicándose el punto donde está el error y su naturaleza. Algo parecido ocurrirá si intenta abrir el documento en Internet Explorer.

Existen otras normas adicionales para que un documento esté bien formado, como por ejemplo la obligatoriedad de que los valores de las propiedades estén entrecomillados. Conocerá estas normas a medida que vaya introduciéndose en el uso de XML.

Como puede observar en el ejemplo de HTML mostrado anteriormente, para que éste sea válido no es preciso cumplir todas las normas. La marca `<p>`, por ejemplo, no cuenta con sus correspondientes cierres: `</p>`, y no por ello el cliente web emite error alguno.

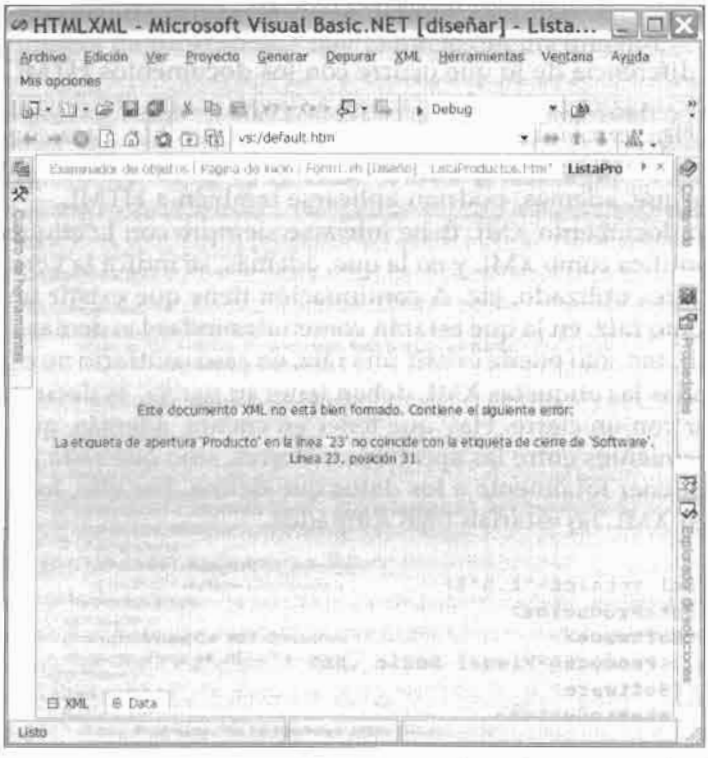


Figura 19.6. El analizador nos indica dónde ha encontrado el error

Documentos XML válidos

Que un documento XML esté bien formado significa que sigue las normas de codificación XML, pero ello no implica que sea un documento válido. El ejemplo mostrado en la figura 19.5, por ejemplo, es un documento XML bien formado, pero a juzgar por su estructura posiblemente no sea válido. ¿Por qué? Observe que algunos productos cuentan con datos de detalle, mientras que otros no. Lo lógico sería que todos siguiesen la misma estructura.

Si no se cuenta con una definición previa de la estructura del documento XML, lo que se conoce como una DTD o un esquema, es posible incurrir en fallos que harían que el documento no fuese válido, a pesar de que sí estuviese bien formado. Puede realizar la siguiente prueba: partiendo del ejemplo de la figura 19.5 añada a la lista de distribuidores del primer producto una pareja `<Editorial>xxx</Editorial>`. Al visualizar el documento podrá ver que no se genera error alguno, ya que el documento está bien formado. No obstante, no es lógico que en el interior de una lista de distribuidores de un producto, que además no es un libro, exista una entrada correspondiente a una editorial.

La única forma de evitar este tipo de fallos consiste en utilizar una DTD o un esquema, que no es más que un documento XML en el que se indica cuál debe ser la estructura de los documentos que se ajusten a dicha definición. Puede disponerse de una DTD o un esquema de dos formas: creándolo uno mismo o

adquiriéndolo de terceros. Actualmente, y como se indicaba al inicio del capítulo, existen decenas de empresas que, de forma conjunta, están creando esquemas para los tipos de negocio más habituales. Esto contribuirá a que existan unos esquemas estándar para determinadas tareas como, por ejemplo, la gestión de pedidos entre empresas.

Como conclusión de lo explicado en éste y el punto anterior, resumir que un documento XML debe estar bien formado, lo cual no implica que sea un documento válido. Para validar un documento XML es necesario disponer de una DTD o un esquema, que podemos crear nosotros mismos si no existe una que se ajuste a nuestras necesidades. Un documento XML válido siempre será un documento bien formado.

Analizadores de XML

Un documento XML es, como ya ha podido ver, un archivo de texto en el que se utiliza un cierto conjunto de caracteres. Para extraer la información contenida en dicho documento, adaptándola a la estructura que puede deducirse de él mismo, se utiliza un analizador o *parser*. Dada la relativa simplicidad de los documentos XML, escribir un analizador no es una tarea compleja en exceso. No obstante, no tiene por qué escribir su propio analizador ya que hay múltiples disponibles para varias plataformas y escritos en distintos lenguajes.

Los analizadores XML se agrupan en dos categorías: validantes y no validantes. Los analizadores no validantes, que son los más sencillos, tan sólo comprueban que el documento XML esté bien formado. Los validantes, además, comprueban si el documento se ajusta a la estructura definida en la DTD o esquema correspondiente, comunicando si el documento es o no válido.

Una vez que el documento se ha analizado y comprobado, existen distintos medios para acceder a la información, ya sea obteniéndola o modificándola. Los dos métodos más conocidos son SAX (*Simple API for XML*) y DOM (*Document Object Model*). Mediante SAX es posible manipular los datos del documento XML de una forma bastante simple, como su propio nombre indica, utilizando manejadores y llamando a métodos. DOM, por el contrario, es un modelo jerárquico de objetos generado a partir de la estructura del documento XML, siendo mucho más flexible y potente.

Trabajando con Visual Basic .NET en un sistema Windows actual, como ya ha podido ver, basta con usar el editor de XML para escribir el código y la página **Data** para acceder a los datos previa comprobación del documento. Usando el menú emergente asociado al documento XML, en la ventana Explorador de soluciones, puede ejecutar la opción Generar y examinar para validar el documento y verlo jerárquicamente, como en Internet Explorer, pero sin salir de Visual Basic .NET (véase figura 19.7).

Validación de documentos

Un documento XML, según se ha dicho antes, puede estar bien formado y, sin embargo, eso no implica que sea válido. En los siguientes puntos explicamos

cómo usar una DTD para validar documentos, introduciendo también nuevos conceptos como los atributos.

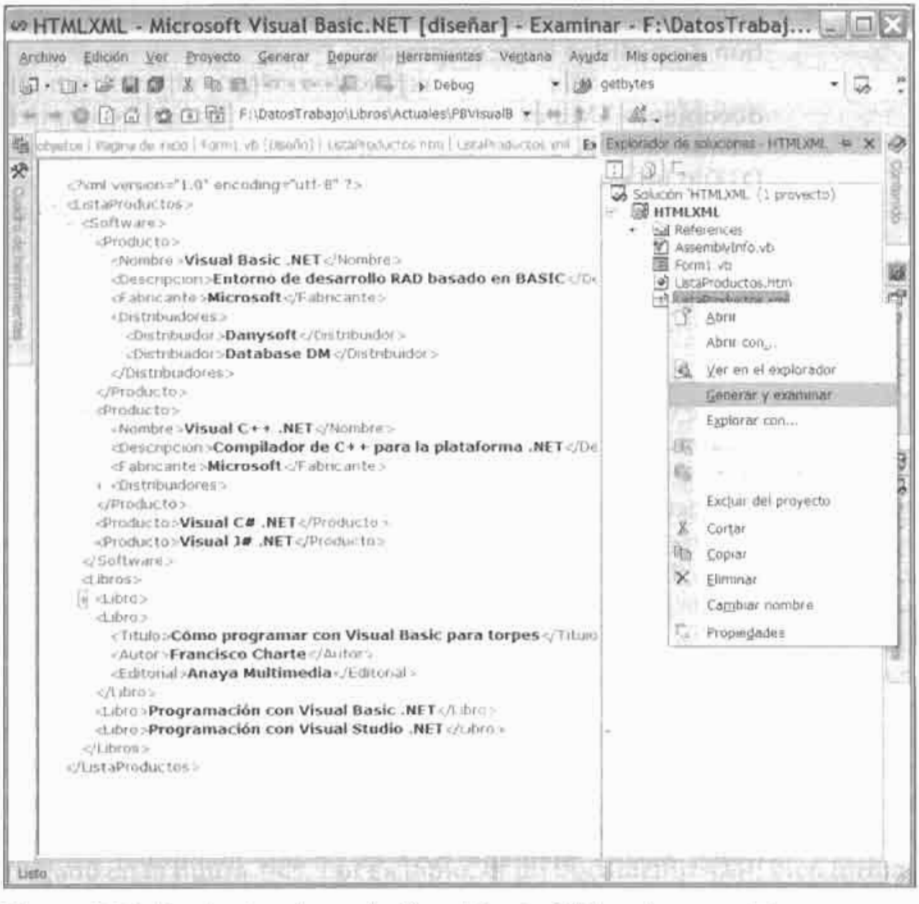


Figura 19.7. Desde el entorno de Visual Basic .NET podemos validar los documentos XML

Nota

Desde el surgimiento de XML el estándar para definir su estructura ha sido el uso de las DTD, lo cual hace que existan muchas definiciones de este tipo. Actualmente, sin embargo, el estándar XSD hace mucho más fácil la creación de esquemas XML. Conocerá básicamente XSD más adelante en este mismo capítulo.

En los puntos previos se ha indicado qué era un documento XML bien formado y qué un documento válido, haciéndose hincapié en las diferencias entre ambos términos. También se ha visto que existen analizadores XML validantes

y no validantes. Los primeros comprueban que el documento esté bien formado y, además, que sea válido. Los segundos, por el contrario, tan sólo verifican si el documento está bien formado.

Aunque es posible verificar manualmente la corrección de un documento, no se trata desde luego de un método práctico, sobre todo cuando se trata de documentos extensos y, además, elaborados por distintas aplicaciones o usuarios. Lógicamente, es mucho más cómodo y eficiente utilizar un analizador validante. Éste, no obstante, necesita conocer las reglas que rigen el documento, de lo contrario no sabrá si es o no válido.

El método utilizado para facilitar las reglas al analizador consiste en escribir una definición de tipo de documento, conocida abreviadamente como DTD. No es necesario escribir una DTD nueva para cada documento XML. De hecho, uno de los mayores objetivos actuales de muchas empresas es crear DTDs estándar que puedan ser usadas para el intercambio de documentos en formato XML. Una DTD generalmente se almacena en un archivo independiente, aunque también puede facilitarse integrada en el propio documento XML.

¿Qué es una DTD?

Físicamente, una DTD es un documento con estructura similar a un documento XML. El contenido, sin embargo, no son datos propiamente dichos, como en el caso de XML, sino indicaciones acerca de la estructura de un determinado tipo de archivo.

A pesar de que una DTD puede parecer, sobre todo al principio, relativamente compleja, en su formato más básico no es más que una enumeración de los elementos que podrán existir en un documento XML, indicando el tipo de cada uno de ellos. Cada elemento se describe en el interior de una etiqueta `<!ELEMENT>`, en la cual se incluirá el nombre del elemento y, entre paréntesis, su tipo o la lista de elementos que podrá contener.

Como es obvio, antes de poder escribir una DTD es preciso un análisis y, posiblemente, una plasmación gráfica de la estructura del documento que quiere definirse. Partiendo de este trabajo, escribir la DTD no será una tarea especialmente compleja.

En lugar de describir teóricamente cuál sería la estructura de una DTD, definición que puede encontrar en cualquier momento en la especificación del estándar, vamos a ver en la práctica cómo crear una DTD. Ésta nos servirá para validar el documento XML utilizado anteriormente como ejemplo, documento que contenía una lista de productos.

Análisis de la estructura del documento

El ejemplo estará basado en el documento XML usado anteriormente, conteniendo una lista de productos de un hipotético comercio. Como puede verse en la figura 19.5, existe un nodo raíz, `<ListaProductos>`, que contiene a otras dos: `<Software>` y `<Libros>`. Ambas sirven como contenedores de varias etiquetas, `<Producto>` en el primer caso y `<Libro>` en el segundo.

Algunas etiquetas `<Producto>` contienen directamente un dato en su interior: el nombre del producto. Otras sirven como contenedor de las etiquetas

<Nombre>, <Descripcion>, <Fabricante> y <Distribuidores>, en el caso de los productos de software, o <Titulo>, <Autor> y <Editorial> si el producto es un libro. Validar un documento de este tipo resultaría relativamente complejo, puesto que las etiquetas <Producto> y <Libro> no siguen un patrón fijo de comportamiento.

Tras remodelar la jerarquía de elementos y pensar la relación existente entre ellos, una estructura mucho más lógica podría ser la mostrada en la figura 19.8. Sigue existiendo un elemento raíz contenedor de otros dos, pero los problemas que existían con los elementos <Producto> y <Libro> desaparecen por completo. Esta representación gráfica y parcial de la estructura nos servirá para escribir la DTD que, a su vez, se utilizará para validar el documento.

Figura 19.8. Estructura de nuestros documentos XML

No hace falta decir que el documento XML del ejemplo que estamos utilizando, según se puede ver en la figura 19.5, no cumple con la estructura de la figura 19.8. Esto significa que, tras escribir la DTD y aplicarla, el analizador nos comunicará que no se trata de un documento XML válido, a pesar de que esté bien formado.

Elaboración de la DTD

Teniendo a la vista la figura 19.8, veamos cómo podemos elaborar la definición de tipo de documento para las listas de productos que gestionarán nues

tras aplicaciones. Como se ha indicado antes, cada uno de los elementos que aparecen en la mencionada figura se traducirá en una etiqueta `<!ELEMENT>`, en la cual se indicará el contenido que puede tener el elemento.

El primer elemento, actuando como raíz de todos los demás, es `ListaProductos`. Éste contendrá en su interior dos subelementos: `Software` y `Libros`. Observe el término *contendrá*, casi imperativo. No se ha dicho *podrá contener*, en cuyo caso los subelementos anteriores podrían o no existir en la lista de productos. Fíjese también en la imposibilidad de que se repitan los subelementos. `ListaProductos` tan sólo contendrá en su interior dos elementos, los ya indicados, ni uno más ni uno menos.

La etiqueta DTD con la que se describiría el elemento `ListaProductos` sería `<!ELEMENT ListaProductos (Software, Libros)>`. Tras la marca de proceso `!ELEMENT` se indica el nombre de la etiqueta raíz y, entre paréntesis, el nombre de las etiquetas que contendrá.

A diferencia de `ListaProductos`, los elementos `Software` y `Libros` pueden contener un número indeterminado de subelementos aunque, eso sí, todos ellos con la misma estructura. Dentro de un elemento `Software` existirán uno o más `Producto`, mientras que en el interior de un `Libros` el elemento repetitivo sería `Libro`. La correspondiente entrada en la DTD, por ejemplo en el caso del elemento `Software`, sería `<!ELEMENT Software (Producto+)>`. Observe el símbolo + dispuesto detrás de `Producto`, él es el que especifica que el elemento se repetirá tantas veces como sea necesario.

La definición DTD de un `Producto` o un `Libro` es bastante sencilla. Basándonos en la definición anterior de `ListaProductos`, lo único que habría que hacer sería enumerar los elementos que habría en su interior, indicando la posible repetición.

En el último nivel se encuentran los elementos que contienen datos, como `Descripcion` o `Autor`. Éstas etiquetas no servirán como contenedoras de otras, sino que contendrán un texto. Este tipo de dato se identifica en una DTD como `#PCDATA`. Así, la definición del elemento `Titulo`, por ejemplo, quedaría como `<!ELEMENT Titulo (#PCDATA)>`.

Con todo, la DTD completa para nuestro documento sería la mostrada a continuación. El sangrado no es algo obligatorio, pero ayuda a identificar los distintos niveles de contención de unos elementos en otros.

```
<!ELEMENT ListaProductos (Software, Libros)>
 <!ELEMENT Software (Producto+)>
   <!ELEMENT Producto (Nombre, Descripcion, Fabricante,
            Distribuidores)>
      <!ELEMENT Nombre (#PCDATA)>
      <!ELEMENT Descripcion (#PCDATA)>
      <!ELEMENT Fabricante (#PCDATA)>
      <!ELEMENT Distribuidores (Distribuidor+)>
         <!ELEMENT Distribuidor (#PCDATA)>
 <!ELEMENT Libros (Libro+)>
   <!ELEMENT Libro (Titulo, Autor, Editorial)>
      <!ELEMENT Titulo (#PCDATA)>
      <!ELEMENT Autor (#PCDATA)>
      <!ELEMENT Editorial (#PCDATA)>
```

Este texto lo almacenaríamos en un documento con extensión DTD que, generalmente, estará alojado en el mismo servidor desde el que se obtenga el documento XML.

Cómo aplicar una DTD a un documento XML

Una definición de tipo de documento no es útil *per sé*, aunque puede usarse como referencia para preparar los documentos XML. La finalidad de una DTD, tal como ya se ha indicado previamente, no es otra que servir a un analizador validante como referencia, facilitando la revisión y validación de cualquier documento XML que se adapte a dicha DTD. Los documentos XML basados en una cierta DTD tienen dos opciones a la hora de indicarlo al analizador: incluir la DTD completa en el propio documento XML o hacer referencia a una DTD externa. El primer caso es útil cuando el propio documento XML es una singularidad, que no va a repetirse y, consecuentemente, consta de una DTD no válida en otros casos. El segundo, mucho más común, es aplicable siempre que la DTD vaya a utilizarse en múltiples documentos. Manteniendo en éstos una referencia externa, se evitará la repetición de la DTD en cada documento.

La instrucción de proceso para especificar el tipo de documento, en el interior del documento XML, es `<!DOCTYPE>`, que irá seguida de un identificador. Tras éste pueden existir unos corchetes, en cuyo interior estaría contenida la DTD, o bien una referencia a la DTD externa. En el ejemplo siguiente puede ver un fragmento del documento XML, conteniendo la DTD completa.

```
<?xml version="1.0"?>
<!DOCTYPE ListaProductos [
<!ELEMENT ListaProductos (Software, Libros)>
 <!ELEMENT Software (Producto+)>
   <!ELEMENT Producto (Nombre, Descripcion, Fabricante,
            Distribuidores)>
      <!ELEMENT Nombre (#PCDATA)>
      <!ELEMENT Descripcion (#PCDATA)>
      <!ELEMENT Fabricante (#PCDATA)>
      <!ELEMENT Distribuidores (Distribuidor+)>
         <!ELEMENT Distribuidor (#PCDATA)>
 <!ELEMENT Libros (Libro+)>
   <!ELEMENT Libro (Titulo, Autor, Editorial)>
      <!ELEMENT Titulo (#PCDATA)>
      <!ELEMENT Autor (#PCDATA)>
      <!ELEMENT Editorial (#PCDATA)>
]>
<ListaProductos>
  <Software>
    <Producto>
       <Nombre>Visual Basic .NET</Nombre>
       <Descripcion>Entorno de desarrollo RAD basado
            en BASIC</Descripcion>
      ...
```

A continuación se muestra el mismo fragmento, pero en este caso con una referencia a la DTD externa. Ésta se ha almacenado en un archivo llamado `ListaProductos.dtd`, que contiene la definición mostrada anteriormente.

```
<?xml version="1.0"?>
<!DOCTYPE ListaProductos SYSTEM "ListaProductos.dtd">
<ListaProductos>
  <Software>
    <Producto>
       <Nombre>Visual Basic .NET</Nombre>
       <Descripcion>Entorno de desarrollo RAD basado
            en BASIC</Descripcion>
      ...
```

En cualquiera de los dos casos, el documento XML ya cuenta con una DTD en la que se especifica la estructura que debería tener ese documento. Un analizador no validante no usaría dicha información para nada, por lo que el documento mostrado anteriormente en la figura 19.5 seguiría siendo correcto, al estar bien formado. Un analizador validante, por el contrario, generaría un error, por ejemplo al encontrar que una etiqueta `<Producto>` contiene directamente el nombre del producto, en vez de contar con las etiquetas `<Nombre>`, `<Fabricante>`, etc.

Validación del documento

Tras añadir la DTD, ya sea en el interior del documento XML o haciendo referencia a ella, habrá que proceder a la validación. Para ello puede utilizarse un analizador validante, de los cuales hay bastantes disponibles en Internet. También puede, simplemente, abrir el documento XML con algún editor o visor de este tipo de documentos.

A pesar de que no disponga de ningún analizador ni editor, si tiene instalado en su sistema la última versión de Internet Explorer ya cuenta con un analizador validante. Éste puede ser usado desde cualquier herramienta de desarrollo, así como desde un *script*. Aprovechando la existencia de WSH (*Windows Scripting Host*) en Windows, que permite la ejecución directa de guiones en VBScript, podríamos servirnos de un *script* como el mostrado a continuación.

```
Set DocXML = CreateObject("Microsoft.XMLDOM")

DocXML.validateOnParse = True

If Not DocXML.Load("ListaProductos.xml") Then

   MsgBox "Hay errores en el documento " & DocXML.url
Else
   MsgBox "El documento " & DocXML.url & " es correcto"
End If
```

Nota

Puede añadir un archivo VBScript al proyecto que tenga abierto usando la *opción adecuada de la ventana Agregar nuevo elemento. Usando la opción* Generar y examinar del Explorador de soluciones, anteriormente indicada.

y pulsando el botón **Abrir** de la ventana que aparece ejecutaremos el guión y veremos el resultado.

El analizador XML incorporado en Internet Explorer es un componente ActiveX, que puede ser usado prácticamente desde cualquier aplicación Windows. En el citado guión, lo que hacemos es crear una copia de ese componente, dando valor a una propiedad con la que se indica que deseamos validar el documento, además de comprobar si está bien formado. A continuación, utilizamos el método `Load()` para abrir el documento, un documento que sabemos que existe en el disco y que no habrá problemas en recuperar. Si el valor devuelto por dicho método es `False`, algo que comprobamos con el operador `Not`, indicará que el documento no es válido. La validez o no del documento se comunica mediante un mensaje.

Tras guardar el guión en un archivo, por ejemplo con el nombre `Validacion.vbs`, bastará con hacer doble clic sobre él desde el Explorador de Windows para ejecutarlo. El resultado será un mensaje como el de la figura 19.9. El documento XML no es válido, puesto que su estructura no cumple con la definición de la DTD.

Figura 19.9. Nuestro *script* nos indica que hay errores en el documento XML

Tendriamos que hacer algunos cambios, dejando el documento tal y como se muestra en el listado siguiente. Al efectuar la validación, ejecutando de nuevo el guión VBScript, el mensaje obtenido ahora sería el de la figura 19.10. El documento sí es correcto y válido, ya que se ajusta a las directrices de la DTD.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE ListaProductos SYSTEM "ListaProductos.dtd">
<ListaProductos>
  <Software>
    <Producto>
      <Nombre>Visual Basic .NET</Nombre>
      <Descripcion>Entorno de desarrollo RAD basado
            en BASIC</Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>Danysoft</Distribuidor>
        <Distribuidor>Database DM</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual C++ .NET</Nombre>
      <Descripcion>Compilador de C++ para la plataforma
            .NET</Descripcion>
      <Fabricante>Microsoft</Fabricante>
```

```
      <Distribuidores>
        <Distribuidor>DanySoft</Distribuidor>
        <Distribuidor>Action</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual C# .NET</Nombre>
      <Descripcion>Nuevo lenguaje orientado
            a componentes</Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>Danysoft</Distribuidor>
        <Distribuidor>Database DM</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual J# .NET</Nombre>
      <Descripcion>Lenguaje Java para la plataforma
            .NET</Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>En preparacion</Distribuidor>
        <Distribuidor>En preparac'on</Distribuidor>
      </Distribuidores>
    </Producto>
  </Software>
  <Libros>
    <Libro>
      <Titulo>Programacion en Windows 2000</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>
      <Titulo>Como programar con Visual Basic para torpes</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>
      <Titulo>Programación con Visual Basic .NET</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>
      <Titulo>Programación con Visual Studio .NET</Titulo>
      <Autor>Jorge Serrano y Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
  </Libros>
</ListaProductos>
```

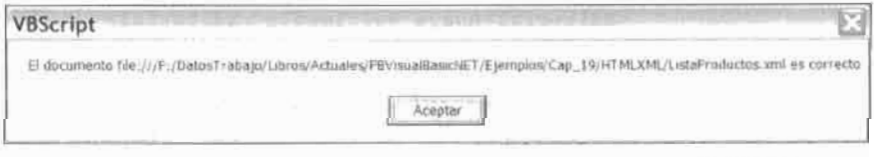


Figura 19.10. Tras los cambios nuestro documento XML sí es válido

Atributos o propiedades

Además de otros elementos y datos, cada una de las etiquetas de un documento XML puede contar también con atributos. Éstos pueden ser considerados como propiedades del elemento, que definen alguna característica. En el ejemplo que estamos utilizando, una propiedad de un producto software podría ser el tipo de máquina al que va destinado: PC, Mac, etc. Para los libros, podría existir un atributo que indicase su categoría: ofimática, programación, etc.

En HTML existen marcas con atributos. La marca `<a>`, por ejemplo, cuenta con un atributo llamado `href` cuyo valor determina el URI del sitio web al que apunta un enlace. La marca `<font>`, por mencionar otro caso conocido, cuenta con atributos como `face` y `size` para establecer el tipo de letra y su tamaño. En todo caso, el formato de las marcas con atributos es siempre el mismo:

```
<marca atributo="valor" atributo="valor">
```

De forma análoga, en XML podrían introducirse etiquetas con atributos, como en el fragmento siguiente. El único inconveniente es que, en principio, a los atributos `Tipo` y `Categoria` podrían asignársele cualquier valor. Como sería de esperar, es posible especificar qué valores son los válidos mediante las entradas apropiadas en la DTD.

```
<ListaProductos>
  <Software>
    <Producto Tipo="Mac">
        <Nombre>VisualAge C++ 4</Nombre>
      <Descripcion>Entorno de desarrollo Java</Descripcion>
        <Fabricante>IBM</Fabricante>
      <Distribuidores>
         <Distribuidor>IBM</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto Tipo="PC">
      <Nombre>Visual Caf&#233; Java 3</Nombre>
      <Descripcion>Entorno de desarrollo Java</Descripcion>
      <Fabricante>Symantec</Fabricante>
      <Distribuidores>
         <Distribuidor>Symantec</Distribuidor>
      </Distribuidores>
    </Producto>
  </Software>
  <Libros>
    <Libro Categoria="Programacion">
       <Titulo>Programaci&#243;n en Windows 2000</Titulo>
       <Autor>Francisco Charte Ojeda</Autor>
       <Editorial>Anaya Multimedia</Editorial>
    </Libro>
```

Si para definir el contenido de un elemento se utiliza la etiqueta `<!ELEMENT>`, para hacer lo propio con los atributos existe la etiqueta `<!ATTLIST>`. Ésta especificará el nombre del atributo, así como los posibles valores de éste en caso

de que sea un conjunto limitado. En principio, la existencia de un atributo en la DTD no implica que en el documento XML dicho atributo se incluya siempre. Si deseamos que esto sea así, que el atributo sea obligatorio, tan sólo hay que finalizar la declaración con la palabra #REQUIRED.

En el listado siguiente puede ver una segunda versión de la DTD original, en este caso incluyendo la definición de los atributos posibles. Puede aplicar esta nueva DTD a una versión modificada del documento XML anterior, en el que se incluyesen los atributos necesarios. El mismo guión de validación le serviría también para comprobar el documento.

```
<!ELEMENT ListaProductos (Software, Libros)>
 <!ELEMENT Software (Producto+)>
   <!ELEMENT Producto (Nombre, Descripcion, Fabricante,
            Distribuidores)>
   <!ATTLIST Producto Tipo (PC | Mac | Consola) #REQUIRED>
      <!ELEMENT Nombre (#PCDATA)>
      <!ELEMENT Descripcion (#PCDATA)>
      <!ELEMENT Fabricante (#PCDATA)>
      <!ELEMENT Distribuidores (Distribuidor+)>
         <!ELEMENT Distribuidor (#PCDATA)>
 <!ELEMENT Libros (Libro+)>
   <!ELEMENT Libro (Titulo, Autor, Editorial)>
   <!ATTLIST Libro Categoria (Sistema | Ofimatica | Programacion)
            #REQUIRED>
      <!ELEMENT Titulo (#PCDATA)>
      <!ELEMENT Autor (#PCDATA)>
      <!ELEMENT Editorial (#PCDATA)>
```

Introducción a XSL

Ya conocemos la estructura de un documento XML, lo que es una DTD y cómo validar un documento. La visualización de los documentos, hasta ahora, siempre se ha efectuado directamente en el editor de Visual Basic .NET o en Internet Explorer en forma de árbol jerárquico. Internet Explorer incorpora no sólo un analizador XML, sino también una hoja de estilo XSL por defecto, que muestra los documentos de este tipo en forma de árbol jerárquico.

La visualización directa del código XML no es lo habitual, ya que el formato puede no ser siempre el más apropiado. Mediante XSL es posible convertir un documento XML en un documento HTML, dando formato a los datos según interese en cada caso.

Los documentos XML pueden ser transformados en HTML o cualquier otro formato gracias a la existencia de XSL. En los siguientes puntos vamos a conocer las bases de XSL.

Transformación de documentos

Los documentos XML son autodescriptivos pero, a pesar de ello, su formato no es el más apropiado para generar informes, ya sean éstos impresos o en

forma de documentos web. Mediante XSL, como veremos a continuación, es posible generar documentos HTML a partir de documentos XML, aplicando el formato que se desee.

En un principio puede parecer que XSL es un lenguaje para aplicar hojas de estilo a los documentos XML, de manera similar a CSS (*Cascading Style Sheets*). La realidad, sin embargo, es que XSL es mucho más potente, facilitando la selección, filtrado y ordenación de los datos, aparte de aplicarles un determinado formato. Por ello, y a pesar de su nombre, XSL puede considerarse más un lenguaje de transformación de documentos, desde formato XML a otros formatos. De hecho, XSL ha sido la base de XSLT que, aunque muy básicamente, conocerá más adelante en este mismo capítulo.

Al igual que XML, XSL es un lenguaje que se utiliza para crear documentos. Estos siguen las reglas de sintaxis de XML. Dicho de otra forma, un documento XSL es un documento XML, aunque en él se utilizan elementos predefinidos que permiten manipular la información. Todas las etiquetas en un documento XSL, al igual que en XML, deben contar con sus respectivos cierres.

Lo habitual, al igual que ocurre con las DTD que hemos conocido anteriormente, es que los documentos XSL se almacenen en archivos independientes, siendo referenciados desde el documento XML. De esta forma, en la cabecera de un documento XML se indicaría cuál es la DTD a la que debería ajustarse y el XSL necesario para transformarlo.

XSL básico

Más que entrar en una definición más o menos formal de qué es XSL y cuál es su sintaxis, información que podrá encontrar en `http://www.w3.org/TR/WD-xsl`, vamos a ver con unos ejemplos sencillos qué podemos conseguir con XSL. Nuestro primer objetivo será generar un documento HTML a partir del documento XML utilizado como ejemplo en puntos previos, un documento conteniendo información relativa a productos de software y libros.

La primera línea de un documento XSL es la conocida marca `<?xml version="1.0" ?>`, con la que se inicia todo documento XML. Para diferenciar al documento XSL de cualquier otro XML, y poder así usar elementos de manipulación de datos en lugar de introducir dichos datos, habrá que añadir la etiqueta `xsl:stylesheet`, que identifica al documento como una hoja de estilo. Dicha marca cuenta con un parámetro, llamado `xmlns` (*XML NameSpace*), que se utiliza para definir un espacio o ámbito de nombres. Tras dos puntos se facilitará el nombre de dicho espacio, mientras que como parámetro se entregará un URL, concretamente el URI en el que se define XSL.

El inicio de cualquier hoja de estilo XSL, por tanto, estará compuesto por las dos líneas siguientes:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl= "http://www.w3.org/TR/WD-xsl" >
```

La etiqueta `xsl:stylesheet` no es de proceso, por lo que deberá contar con su respectiva marca de cierre. En este caso se ha asignado al espacio de

nombres el nombre `xsl`, que es lo habitual. Podría, no obstante, asignarse cualquier otro nombre. En cualquier caso, hay que tener en cuenta que los elementos definidos en la `www.w3.org/TR/WD-xsl`, usados en el resto del documento, deberán ir precedidos del identificador del espacio de nombres. Para aplicar una plantilla de estilo, por ejemplo, se utiliza el elemento `template`. Si el documento se inicia con el fragmento anterior, dicho elemento aparecería como `xsl:template`.

Una hoja de estilo compuesta tan sólo de la cabecera anterior, junto con la etiqueta de cierre, no es que sea precisamente útil, pero ya podemos almacenarla en un archivo con extensión `XSL` y hacer referencia a ella desde un documento XML. Dicha referencia se efectúa mediante una marca de proceso como la siguiente:

```
<?xml-stylesheet type="text/xsl" href="EstiloSimple.xsl"?>
```

Tal y como puede verse, la marca `xml-stylesheet` cuenta con dos parámetros: `type` y `href`. Con el primero se indica el tipo de hoja de estilo, que está codificada en XSL en este caso, y con el segundo se facilita el URL en que se encuentra.

Si el documento XML es obtenido a través de la red desde un determinado servidor, normalmente su manipulación está restringida a hojas de estilo recuperadas del mismo servidor y no desde cualquier otro punto, sea local o bien remoto.

Cómo aplicar plantillas

En el interior de la hoja de estilo se aplicarán plantillas a determinadas partes del documento XML. Estas plantillas establecen el formato de salida de la información, por ejemplo generando las etiquetas HTML asociadas a cada elemento del documento XML.

El elemento XSL a utilizar es `template`, pudiendo aplicarse a todo el documento o a una parte de él. Mediante el parámetro `match` se establecerá el conjunto de datos al que se aplicará la plantilla. El argumento de dicho parámetro será `/`, para seleccionar todo el documento, o bien el nombre del elemento al que desee aplicarse la plantilla, por ejemplo `Libro`.

Entre la marca de apertura y la de cierre de la plantilla podremos introducir todos los elementos que deseemos aplicar, tanto de formato como de proceso. Los de formato serán simples etiquetas HTML, por ejemplo, mientras que los de proceso serán otros elementos, conoceremos algunos en un momento, que nos permitirán seleccionar datos del documento XML e introducirlos en la plantilla.

Observe el código siguiente, correspondiente a una hoja de estilo XSL almacenada en el archivo `PlantillaGeneral.XSL`. Tan sólo contiene una plantilla que se aplicará al documento completo, compuesta de varias etiquetas HTML. Observe que los elementos XSL están precedidos del prefijo `xsl` y dos puntos. Esto se debe a que en la marca `stylesheet` se ha llamado `xsl` al

espacio de nombres. Si lo hubiésemos llamado de otra forma, habría que sustituir el prefijo `xsl` por el que correspondiese.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >

<xsl:template match="/" >

 <H1>Lista de productos</H1>

 <H2>Software</H2>

 <H2>Libros</H2>

</xsl:template>

</xsl:stylesheet>
```

La hoja de estilo por sí sola no es útil, lógicamente. Hay que aplicarla al documento XML, para lo cual habrá que incluir en éste la correspondiente marca de proceso `?xml-stylesheet`, tal y como puede apreciarse en el siguiente fragmento. Vamos a seguir usando en nuestros ejemplos el mismo documento de artículos utilizado con anterioridad.

```
<?xml version="1.0"?>
<!DOCTYPE ListaProductos SYSTEM "ListaProductos2.dtd">
<?xml-stylesheet type="text/xsl" href="PlantillaGeneral.xsl"?>
<ListaProductos>
  <Software>
    <Producto>
       <Nombre>Visual Basic .NET</Nombre>
        ...
```

Si utilizamos el editor XML de Visual Basic .NET seguiremos obteniendo la misma tabla de datos con los elementos del documento XML, pero abriéndolo en Internet Explorer podremos ver que ya no aparece la lista jerárquica con todos los elementos y datos del documento. En su lugar, como puede ver en la figura 19.11, tan sólo aparecen los títulos correspondientes a las etiquetas HTML introducidas en el interior de la plantilla XSL.

Selección de datos

Una plantilla que genera un documento tan sólo con unos títulos, sin tomar la información almacenada en el documento XML, no es nada útil. Veamos, por tanto, cómo podemos seleccionar los datos de dicho documento, incluyéndolos en el interior de esa plantilla genérica para crear otra nueva, relativamente simple pero más útil.

Puesto que la estructura de todo documento XML suele estar compuesta de patrones repetitivos, en nuestro caso elementos `Producto` y `Libro` que se repiten en todo el documento, en XSL existe un elemento de enumeración llama-

do `for-each`. Este dispone de un parámetro, que se llama `select`, cuyo argumento determinará qué datos se van a elegir. La etiqueta `<xsl:for-each select="ListaProductos/Software/Producto" >`, por ejemplo, elige todas las entradas `Producto` que hay en la rama `Software` de la `ListaProductos`. Como puede deducirse, este elemento XSL lo que hace es crear un bucle en el cual los elementos siguientes, hasta encontrar la etiqueta de cierre, se repiten una vez por cada entrada `Producto`.

Figura 19.11. Resultado que genera la plantilla XSL simple

Con el elemento `for-each`, por lo tanto, podemos determinar a qué datos afectará el bucle. En el interior de éste, no obstante, necesitaremos recuperar información de los apartados que forman cada `Producto`, como el nombre, distribuidor, fabricante, etc. Con este fin se utiliza el elemento `value-of` que, al igual que el anterior, cuenta con un parámetro `select` que servirá para indicar el dato a recuperar.

Usando los elementos `for-each` y `value-of` podemos crear una hoja de estilo como la mostrada a continuación, partiendo de la que habíamos escrito previamente.

Observe que existen dos construcciones `for-each`, una para recorrer los productos de software y otra para los libros. En el interior de cada una de ellas simplemente se introduce el nombre del producto o título del libro, además de unas etiquetas HTML para separar cada elemento del siguiente.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >

<xsl:template match="/" >

 <H1>Lista de productos</H1>

 <H2>Software</H2>
 <xsl:for-each select="ListaProductos/Software/Producto" >
      <P><xsl:value-of select="Nombre" /></P>
 </xsl:for-each>

 <H2>Libros</H2>
 <xsl:for-each select="ListaProductos/Libros/Libro" order-by="Autor">
      <P><xsl:value-of select="Titulo" /></P>
 </xsl:for-each>

</xsl:template>

</xsl:stylesheet>
```

Aplicando esta nueva hoja de estilo al documento XML, para lo cual habrá que sustituir la referencia que hay en éste, ahora obtendremos un resultado similar al de la figura 19.12. El documento es bastante más útil e, incluso, más claro que la visualización directa del árbol jerárquico XML, sobre todo para una persona que desconozca dicho lenguaje.

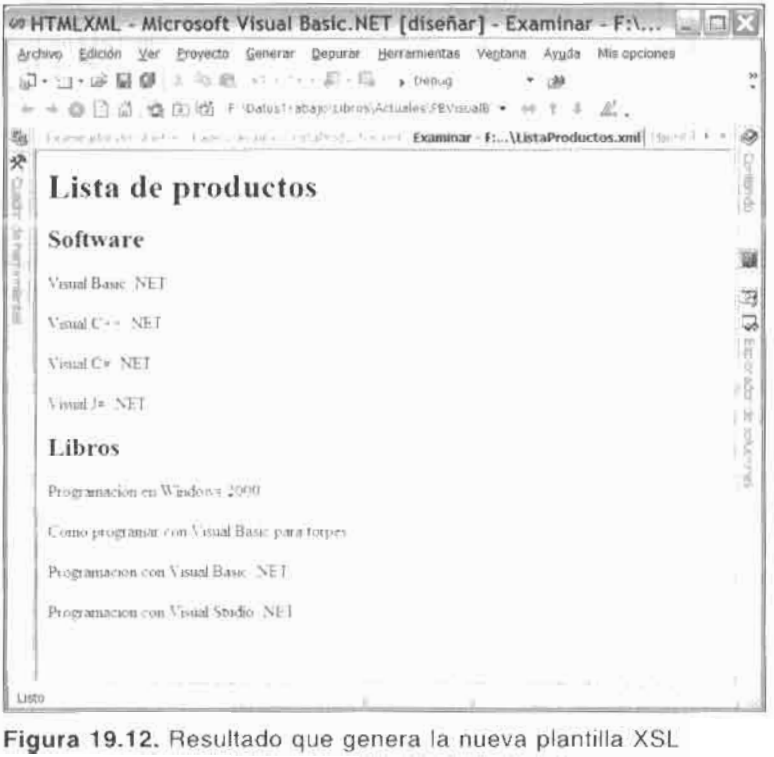


Figura 19.12. Resultado que genera la nueva plantilla XSL

El resultado obtenido sigue siendo simple, pero realmente no necesitamos conocer ningún elemento XSL más para generar documentos mucho más elaborados. En el listado siguiente puede ver otra hoja XSL en la que tan sólo se usan los elementos ya conocidos, pero en la que se ha introducido más formato utilizando etiquetas HTML. Básicamente, se crea una tabla con unos títulos y en las celdillas centrales se muestran algunos datos del documento XML original. El resultado (véase figura 19.13) es mucho mejor que el anterior, válido para una presentación al usuario.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >

<xsl:template match="/" >

 <TABLE STYLE="border:1px solid">
  <TR>
   <TD COLSPAN="2" ALIGN="center" BGCOLOR="silver">
    <H1>Lista de productos</H1>
   </TD>
  </TR>

  <TR>
   <TD COLSPAN="2" ALIGN="center" BGCOLOR="aqua">
    <H2>Software</H2>
   </TD>
  </TR>

  <TR BGCOLOR="ocean">
   <TD ALIGN="center">
    <H4>Nombre</H4>
   </TD>
   <TD ALIGN="center">
    <H4>Fabricante</H4>
   </TD>
  </TR>

  <xsl:for-each select="ListaProductos/Software/Producto" >
   <TR BGCOLOR="ocean">
    <TD><xsl:value-of select="Nombre" /></TD>
    <TD><xsl:value-of select="Fabricante" /></TD>
   </TR>
  </xsl:for-each>

  <TR>
   <TD COLSPAN="2" ALIGN="center" BGCOLOR="aqua">
    <H2>Libros</H2>
   </TD>
  </TR>

  <TR BGCOLOR="ocean">
   <TD ALIGN="center">
    <H4>Titulo</H4>
   </TD>
   <TD ALIGN="center">
    <H4>Autor</H4>
   </TD>
  </TR>
```

```
    <xsl:for-each select="ListaProductos/Libros/Libro"
        order-by="Autor">
     <TR BGCOLOR="ocean">
       <TD><xsl:value-of select="Titulo" /></TD>
       <TD><xsl:value-of select="Autor" /></TD>
     </TR>
    </xsl:for-each>
  </TABLE>
</xsl:template>
</xsl:stylesheet>
```

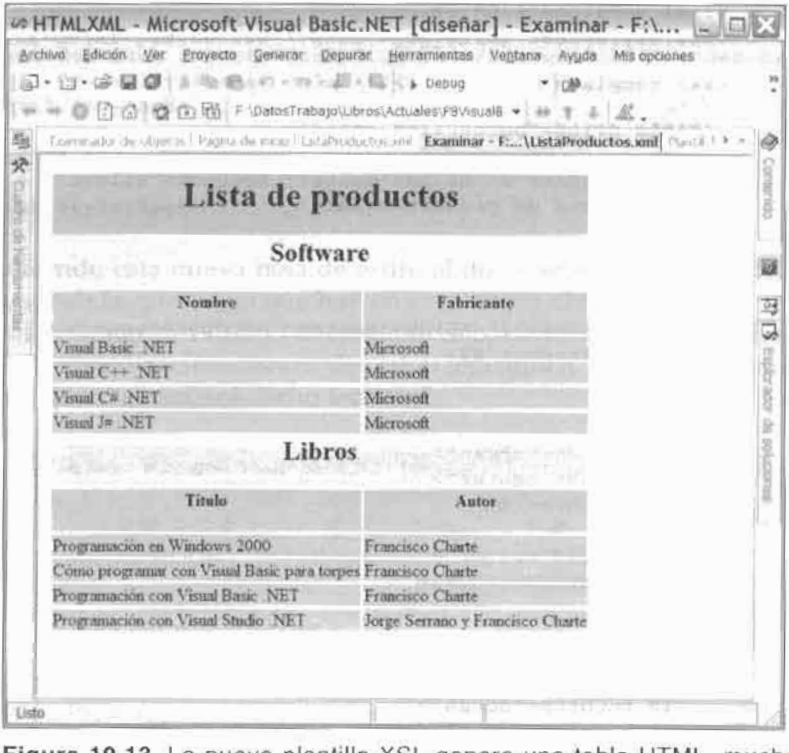


Figura 19.13. La nueva plantilla XSL genera una tabla HTML, mucho más legible

Las posibilidades de XSL

Tan sólo con tres elementos XSL, los explicados `template`, `for-each` y `value-of`, es posible convertir cualquier documento XML en una página HTML tan compleja como se desee. Las posibilidades de XSL, no obstante, no terminan aquí. Hay más elementos de manipulación de datos, así como un complejo y eficiente lenguaje de comparación de patrones que facilita la selección de información y toma de alternativas.

Al seleccionar los conjuntos de datos, con el elemento `for-each`, es posible ordenarlos según el criterio que nos interese. Podríamos, por ejemplo, ordenar los productos por fabricante o bien los libros por autores. Con elementos como

choose, when y otherwise es posible plantear alternativas, o tomar decisiones en el argot de los programadores. Esto puede utilizarse, por ejemplo, para aplicar un formato condicional a los datos. Mediante el elemento eval podríamos evaluar expresiones incluyendo el resultado en el documento a generar.

Además de los elementos propiamente dichos, en XSL existe una serie de métodos predefinidos que facilitan, por ejemplo, la salida de fechas, horas y números, así como la obtención de los índices que identifican a cada uno de los elementos del documento XML.

Introducción a XSD

Previamente, en este mismo capítulo, hemos visto cómo la estructura de un documento XML puede representarse mediante una DTD que, además, puede ser utilizada por el analizador XML para efectuar el proceso de validación. En principio las DTD eran el único mecanismo para documentar la estructura, pero durante meses un grupo de trabajo del W3C se ha centrado en la definición de un nuevo sistema basado en esquemas.

Los esquemas XML son más flexibles que las DTD y se trata de un mecanismo que Microsoft está potenciando en sus aplicaciones. El analizador XML de Microsoft es capaz de trabajar con esquemas que, como en el caso de las DTD, pueden alojarse en el propio documento XML o en un archivo separado al que se referencia desde el primero.

Un esquema XML es en sí mismo un documento XML que, por tanto, puede ser procesado con cualquier analizador XML, transformado con una hoja XSL y mostrado en un cliente como Internet Explorer. A diferencia de una DTD, en un esquema XML los elementos que se definen pueden contar con tipos de datos, es posible extender el esquema con posterioridad y las definiciones pueden anidarse.

El lenguaje XSD (*XML Schema Definition Language*) es mucho más versátil que DTD, contando con tipos de datos básicos, la posibilidad de definir tipos complejos, aplicar restricciones, etc. Además, XSD está basado en XML, mientras que DTD, como ya se ha visto, no se ajusta a dicho estándar.

Un esquema XSD se inicia con la etiqueta <schema> que, habitualmente, suele llevar implícita la declaración de un ámbito con nombre, de forma similar a lo explicado previamente para XSL. Toda la descripción, la definición del esquema, estará contenida entre las marcas <schema> y </schema>. La estructura básica, por tanto, sería ésta:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:definición de elementos>
  </xs:definición de elementos>
</xs:schema>
```

La definición de cada elemento se efectúa con la marca <element>. Ésta irá acompañada como mínimo de dos atributos: Name y Type, indicando el nombre del elemento y su tipo. Los tipos básicos son números enteros y con parte

decimal, cadenas, fechas, etc. Prácticamente se contempla el uso de cualquier tipo de dato existente en los lenguajes actuales.

```
<xs:element name="Titulo" type="xs:string"/>
```

Con la etiqueta anterior, por ejemplo, se define un elemento que tendrá por nombre `Titulo` que contendrá una cadena de caracteres. Vea que se usa el prefijo `xs` como nombre del espacio que asumimos se ha definido previamente.

En caso de que el tipo no sea tan simple como un número o una cadena, será necesaria una descripción indicando los elementos de que se compone. En este caso la etiqueta que nos interesa es `<ComplexType>` que, como verá en el fragmento siguiente, dispone de un atributo `name` que permite indicar el nombre del tipo. En su interior se definirán los elementos, según la sintaxis de la etiqueta `<element>` que acaba de explicarse, y, en caso necesario, también los atributos.

```
<xs:complexType name="LibroType">
  <xs:sequence>
    <xs:element name="Titulo" type="xs:string"/>
    <xs:element name="Autor" type="xs:string"/>
    <xs:element name="Editorial" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="Categoria" type="xs:string"/>
</xs:complexType>
```

Obsérvese que la definición de los elementos está contenida en una marca `<sequence>`. De esta forma se indica que los elementos deben aparecer en ese orden y no en cualquier otro. Fíjese también en que la definición del único atributo existente es idéntica a la de los elementos, simplemente se sustituye el elemento `<element>` por `<attribute>`.

A partir de este momento, podríamos usar `LibroType` como cualquier otro tipo básico, utilizándolo en la definición de nuevos elementos.

En caso de que un elemento deba repetirse, como si fuese una lista o una matriz, aparte del nombre y el tipo habrá que utilizar el atributo `minOccurs`, `maxOccurs` o ambos. En la siguiente definición, por ejemplo, se indica que el tipo `DistribuidoresType` estará compuesto de uno o más nombres de distribuidor. Observe que el valor de `maxOccurs` es la palabra `unbounded`, indicando así que no hay un máximo preestablecido.

```
<xs:complexType name="DistribuidoresType">
  <xs:sequence>
    <xs:element name="Distribuidor" type="xs:string"
          minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

El editor de esquemas XML de Visual Basic .NET

Visual Basic .NET cuenta con un editor de esquemas que podríamos calificar de diseñador puesto que, de una forma casi visual, podemos establecer los

tipos de datos y sus elementos, así como las relaciones que darán lugar a la jerarquía del documento. Para añadir un nuevo esquema XMI. nos serviremos, como es habitual, del cuadro de diálogo Agregar nuevo elemento mostrado en la figura 19.14. Automáticamente se abrirá el diseñador de esquemas en el que, utilizando las opciones del menú emergente, iremos añadiendo elementos, atributos y tipos.

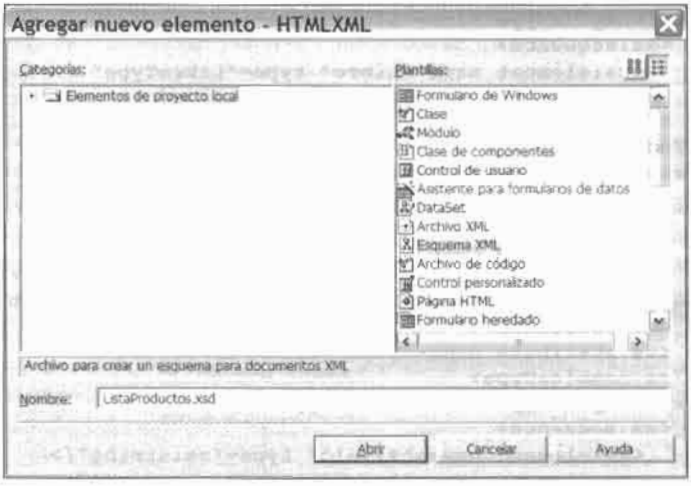


Figura 19.14. Añadimos un nuevo esquema XML

Utilizando este diseñador sería fácil definir los elementos de cada uno de los tipos que contendrá un documento, en nuestro caso esos tipos serían `Software` y `Libros` que, a su vez, contendrían otros tipos. Si ya tenemos un documento XMI. creado, podemos generar automáticamente el esquema XSD usando la opción XML>Crear esquema. Generado el esquema, ése se asociará con el documento XMI. y podremos utilizar la opción XML>Validar datos XML para comprobar su validez. Al ir introduciendo datos, además, los posibles errores se indicarán como es habitual en el editor de código de Visual Basic, con un subrayado de color.

Codificación manual del esquema

Aunque tan sólo se han descrito los conceptos mínimos de XSD, tenemos suficiente información para crear un esquema que describa la estructura de nuestros documentos XML, concretamente el último usado como ejemplo compuesto de una lista de productos, sin necesidad de recurrir al diseñador que incorpora Visual Basic .NET. El esquema sería el siguiente:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ListaProductos" type="ListaProductosType"/>
  <xs:complexType name="ListaProductosType">
    <xs:sequence>
      <xs:element name="Software" type="SoftwareType"/>
```

```
      <xs:element name="Libros" type="LibrosType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="SoftwareType">
    <xs:sequence>
      <xs:element name="Producto" type="xs:string"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LibrosType">
    <xs:sequence>
      <xs:element name="Libro" type="LibroType"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ProductoType">
    <xs:sequence>
      <xs:element name="Nombre" type="xs:string"/>
      <xs:element name="Descripcion" type="xs:string"/>
      <xs:element name="Fabricante" type="xs:string"/>
      <xs:element name="Distribuidores" type="DistribuidoresType"/>
    </xs:sequence>
    <xs:attribute name="Tipo" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="LibroType">
    <xs:sequence>
      <xs:element name="Titulo" type="xs:string"/>
      <xs:element name="Autor" type="xs:string"/>
      <xs:element name="Editorial" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="Categoria" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="DistribuidoresType">
    <xs:sequence>
      <xs:element name="Distribuidor" type="xs:string"
        minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

En este esquema se indica que nuestro documento tendrá un elemento llamado `ListaProductos` que será de tipo `ListaProductosType`. En dicho tipo se especifica que el elemento contará con dos subelementos: `Software` y `Libros`, que deberán aparecer en ese orden. Continuando con el análisis llegaremos hasta los tipos más sencillos, los elementos y atributos que contienen un dato simple como `string`.

Introducción a XSLT y XPath

Aunque no son dos temas que nos interesen demasiado en este momento, lo cierto es que tras haber conocido los fundamentos de XSL no es difícil comprender qué son y para qué se utilizan XSLT (*XSL Transformations*) y XPath. Podríamos decir que ambos son un subconjunto de XSL, o bien que son lenguajes

derivados de XSL. Ambos, XSLT 1.0 y XPath 1.0, son actualmente trabajos en proceso en el W3C.

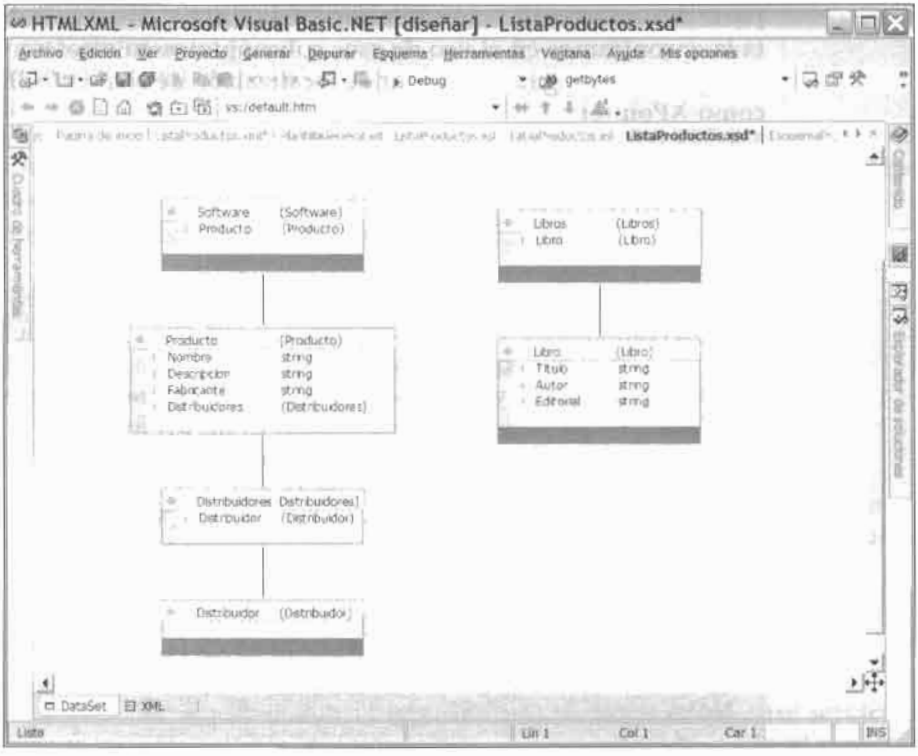


Figura 19.15. Aspecto del diseñador de esquemas durante la creación del esquema para nuestro documento

XPath es un lenguaje de selección de datos contenidos en documentos XML. Usando una analogía, podríamos decir que XPath es a un documento XML lo que SQL a una base de datos. XPath, por lo tanto, es simplemente un lenguaje, siendo necesaria una aplicación o componente que sea capaz de interpretarlo adecuadamente y ejecutarlo, lo mismo que los sistemas RDBMS interpretan y ejecutan las consultas expresadas en SQL.

XSLT es un lenguaje de transformación de datos, utilizando XPath para seleccionar la información de un documento XML y, posteriormente, generar otro distinto, ya sea XML, HTML o en otro formato. En realidad, XSLT es la parte de XSL que ya conocemos, y utilizamos en un ejemplo previo, para generar un documento a partir de otro distinto.

En este contexto, una hoja de estilo XSL es un documento que, utilizando sintaxis XML, combina XSLT, XPath y los objetos de formato XSL para generar unos documentos a partir de otros. Si tomamos la última hoja XSL utilizada como ejemplo anteriormente, los elementos que localizan la información, como `match="/"` o `select="ListaProductos/Software/Producto"`, son parte del lenguaje XPath, mientras que los elementos que manipulan la información

y la convierten, como `template` y `for-each`, pertenecen al lenguaje de transformación XSLT.

¿Qué sentido tiene independizar aspectos que, como XPath y XSLT, forman parte de una especificación anterior y más amplia, como es la de XSL? La respuesta la encontramos en el uso de esos subconjuntos en otros contextos. XPath, por ejemplo, es el lenguaje estándar de localización de elementos XML en estándares como XPointer.

Introducción a SOAP

XML no tiene aplicación sólo como estándar para la creación de documentos intercambiables entre aplicaciones y sistemas, sino que tiene también un gran peso en el nuevo contexto de los servicios web. No en vano, XML es la base del protocolo SOAP y el lenguaje WSDL que trataremos en el siguiente punto.

SOAP (*Simple Object Access Protocol*) es un protocolo RPC (*Remote Procedure Call*) que utiliza XML como lenguaje para describir la acción y HTTP como protocolo de transporte a través de la red. Esto hace que sea compatible con cualquier lenguaje y sistema operativo, ya que XML es un estándar ampliamente aceptado. Al mismo tiempo, utilizando HTTP como protocolo de transporte se evitan los problemas que plantea el uso de otras técnicas, como DCOM o CORBA, en redes donde existen cortafuegos.

Nota

Realmente SOAP puede usarse con otros protocolos de transporte, como SMTP, no está diseñado exclusivamente para utilizarse con HTTP aunque, actualmente, ésta sea la opción más lógica y utilizada.

Mediante SOAP se componen mensajes que van desde el cliente, que efectúa una petición, al servidor, indicándole cuál es la acción o el servicio que desea usar y, en caso necesario, facilitándole los parámetros que fuesen necesarios. El servidor, en respuesta, remitirá otro mensaje al cliente con los resultados.

Estructura de un mensaje SOAP

Los mensajes SOAP son documentos XML que se ajustan a un cierto esquema. Este determina que deberá existir un elemento `Envelope` que contenga todo el mensaje, conteniendo elementos opcionales, como una cabecera, y otros obligatorios, como `Body`, en el que se indicará el servicio al que desea accederse.

No hay diferencia entre un mensaje de solicitud y otro de respuesta, salvo en que utilizan mensajes distintos de HTTP para viajar del cliente al servidor o viceversa. Imaginando que tuviésemos en marcha un servicio de información bibliográfica, un cliente imaginario podría enviar este mensaje:

```
<SOAP-ENV:Envelope>
 <SOAP-ENV:Body>
   <InformacionLibro>
     <ISBN>84-415-0967-0</ISBN>
   </InformacionLibro>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

En el servidor al que se envie este mensaje existirá un componente que lo identificará, tomará el parámetro facilitado y generará una respuesta con la información correspondiente.

El mensaje de respuesta bien podría ser el siguiente:

```
<SOAP-ENV:Envelope>
 <SOAP-ENV:Body>
   <InformacionLibroRespuesta>
     <Titulo>Programación con Visual Basic .NET</Titulo>
   </InformacionLibroRespuesta>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Elementos como `InformacionLibro` e `InformacionLibroRespuesta` estarían definidos en algún esquema, mientras que los elementos `Envelope` y `Body` son parte de la especificación de SOAP.

SOAP y Visual Basic .NET

Visual Basic .NET usa SOAP para permitir la comunicación entre servicios Windows y las aplicaciones que consumen dichos servicios. Todo el proceso de generación de los mensajes, transferencia, recepción e interpretación queda en manos de objetos ya predefinidos en la plataforma .NET, de manera que nosotros nunca tendremos la necesidad de trabajar manualmente con este tipo de información.

Introducción a WSDL y UDDI

Para poder invocar a un servicio web, componiendo adecuadamente el mensaje SOAP para efectuar la solicitud, es necesario contar con una descripción de ese servicio. En esa descripción se enumeran las funciones con que cuenta el servicio, los parámetros que necesitan, las respuestas que generan y los tipos de datos utilizados en todo el proceso. Las empresas ofrecerán esos servicios a través de Internet, y la descripción, lógicamente, estará almacenada en un servidor. Es necesario, por tanto, algún mecanismo que permita localizar la descripción. Aquí es donde entran en escena UDDI y WSDL.

UDDI (*Universal Discovery, Description and Integration Service*) es un servicio de directorio puesto en marcha por empresas como Microsoft e IBM. Su finalidad es servir, a modo de una guía telefónica, para localizar el tipo de servicio

por el que está interesado el cliente. El servicio UDDI le facilitará las referencias necesarias para acceder a la descripción de esos servicios.

La descripción de un servicio web se efectúa mediante WSDL (*Web Service Description Language*), otro lenguaje basado en XML. En un archivo WSDL se combinan elementos XML específicos para la descripción del servicio, por ejemplo los conectores para acceder a él, y esquemas que definen los datos que se enviarán o devolverán las funciones del servicio.

El funcionamiento conjunto de UDDI, WSDL y SOAP queda reflejado en la figura 19.16. La aplicación recurre, en primera instancia, al registro UDDI para localizar el servicio que está buscando. De dicho registro obtiene uno o varios URL, que usa para acceder a los servidores que ofrecen esos servicios en busca de la descripción WSDL. Disponiendo de ésta, la aplicación ya puede componer adecuadamente los mensajes SOAP para consumir esos servicios.

No tenemos que preocuparnos por la generación de la descripción WSDL, ya que ésta es una tarea que realiza automáticamente Visual Basic .NET como podremos experimentar en los puntos siguientes. También hay herramientas de terceros para efectuar ese trabajo.

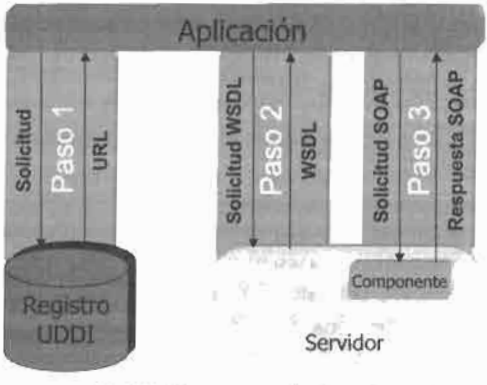


Figura 19.16. Esquema de funcionamiento de una aplicación que consume un servicio web

Creación de un servicio Web

Para poder hacer pruebas con un servicio Web, desde cualquier tipo de aplicación, primero debemos disponer del propio servicio. Por ello comenzaremos creando un servicio simple, dispondrá tan sólo de un método cuya finalidad será devolver una cadena de caracteres con la fecha y hora actuales. Será suficiente, no obstante, para que conozcamos la estructura básica de un servicio Web y podamos, posteriormente, utilizarlo como base para crear otro de mayor complejidad.

Aunque vamos a usar Visual Basic .NET para crear nuestro servicio Web, realmente no necesitaríamos más herramientas que el Bloc de notas para escribir un módulo que sería compilado dinámicamente por parte de ASP.NET.

Los servicios Web son accesibles, generalmente, a través de un servidor Web. El punto de entrada es un archivo con extensión `asmx`. Éste puede contener el código del servicio o bien una referencia a un módulo que contiene dicho código.

Anatomía de un servicio Web

La clase de la cual deben derivarse los servicios web es `WebService`, alojada en el ámbito con nombre `System.Web.Services`. En el interior de esta nueva clase, que heredará los miembros de `WebService`, podemos incluir todos los métodos que necesitemos, ya sean privados, protegidos o públicos, utilizando la sintaxis propia del lenguaje que estemos usando. Ninguno de esos métodos, sin embargo, sería accesible para un consumidor del servicio.

Aparte de listas de parámetros, tipos de retorno y especificadores, un método también puede contar con atributos, según vimos en un capítulo previo. Éstos se introducen entre los símbolos < y > en el caso de Visual Basic .NET, y se colocan, generalmente, delante de la cabecera del método.

Si deseamos que una cierta clase pueda considerarse un servicio web, aparte de derivarla de la clase `WebService` también deberemos definir al menos un método con el atributo `WebMethod`. De esta forma indicaríamos qué métodos de la clase estarán accesibles para los consumidores, simplemente disponiendo `<WebMethod>` delante de la cabecera de los métodos que nos interesen.

Por último, el documento `asmx` deberá iniciarse mediante la directiva `<%@ WebService`, indicándose el lenguaje en el que está escrito el código o, si procede, facilitando el ámbito y nombre de la clase si ésta ya se encuentra precompilada.

El servicio horario

Para crear nuestro servicio Web nos serviremos, una vez más, de los asistentes que hay en la ventana **Nuevo proyecto**, tal y como puede verse en la figura 19.17. Este asistente generará el archivo `asmx` indicando en él que el código del servicio Web se encuentra en el módulo `asmx.vb` que, como puede suponer, sólo contiene código Visual Basic .NET.

En principio se encontrará con un diseñador muy similar al utilizado en un capítulo previo para la creación de formularios Web. Pulse el enlace que le permite acceder al código y observe entonces la definición de la clase, derivada de `System.Web.Services.WebService`. En el constructor hay una llamada al método `InitializeComponent()`, como en otros tipos de aplicación, si bien en este caso dicho método está vacío.

Añada el código siguiente al final de la clase:

```
<WebMethod()> _
Public Function Horas() As String
  Return DateTime.Now.ToString()
End Function
```

Figura 19.17. Iniciamos el desarrollo de nuestro servicio Web

Como puede ver, el método difícilmente podría ser más sencillo. Recuperamos la fecha y hora actuales, de la propiedad `Now` de la clase `DateTime`, convertimos el dato en una cadena y lo devolvemos. Lo único especial es el atributo `<WebMethod>` delante del método.

Sólo con esto ya tiene desarrollado su primer servicio Web, no necesita hacer nada más. Desde cualquier ordenador podría obtenerse la hora del servidor en que se ejecuta el servicio, sin necesidad de escribir ninguna aplicación y sin importar la plataforma.

El módulo `asmx`

Hasta ahora hemos estado trabajando con el módulo que contiene el código del servicio, un módulo que será compilado desde Visual Basic .NET generando un ensamblado y que, por tanto, no habrá que compilar dinámicamente. Aunque en el Explorador de soluciones aparece el módulo `asmx`, si hace doble clic sobre él se encontrará con el módulo de código, el módulo `asmx.vb`. Esto es así porque el módulo `asmx`, que es el punto de entrada al servicio Web, es mantenido internamente por Visual Basic .NET.

Para abrir el módulo `asmx` tendrá que desplegar su menú emergente, pulsando sobre dicho módulo en el Explorador de soluciones con el botón secundario del ratón, y después elegir la opción Abrir con. Aparecerá una ventana, similar a la de la figura 19.18, en la que podremos elegir el tipo de editor a utilizar. Tras pulsar el botón **Abrir** podrá, finalmente, ver el contenido del módulo `asmx`.

El contenido inicial de este módulo es el que puede verse a continuación:

```
<%@ WebService Language="vb"
    Codebehind="Service1.asmx.vb"
    Class="ServicioHorario.Service1" %>
```

Figura 19.18. Seleccionamos el tipo de editor para abrir el módulo `asmx`

Simplemente se indica el lenguaje, módulo y clase donde se encuentran los métodos del servicio Web. En la práctica, podríamos sustituir ese código por el siguiente y olvidarnos del módulo `asmx.vb`, incluyendo toda la lógica en el módulo `asmx`. La compilación, además, sería dinámica.

```
<%@ WebService Language="vb" _
     class="ServicioHorario.ServicioHorario" %>

Imports System
Imports System.Web.Services


 Public Class ServicioHora
   Inherits WebService


  <WebMethod> _
  Public Function Horas() As String

     return DateTime.Now.ToString()
  End Function
 End Class
```

Aunque nuestro proyecto cuenta, aparte de con los mencionados módulos `asmx` y `asmx.vb`, con un archivo de configuración, `web.config`, y algunos elementos más, lo cierto es que un archivo con extensión `asmx` conteniendo el código anterior y colocado en la carpeta raiz del servidor sería suficiente para disponer de un servicio web.

Módulo de descripción del servicio

Si quisiéramos registrar nuestro servicio Web en un directorio, a fin de que cualquiera pudiera hacer uso de él, necesitaríamos un módulo de descripción. En él se encontraría información sobre cómo acceder al servicio y a sus métodos. Conociendo la sintaxis de WSDL, que es el lenguaje en el que debe facilitarse

esta descripción, podríamos crearla a mano. No obstante, ASP.NET es capaz de generar automáticamente la descripción de cualquier servicio Web que hayamos creado con Visual Basic .NET.

Asumiendo que tenemos abierto el servicio horario que acabamos de desarrollar, bastará con ejecutarlo para acceder a una página como la de la figura 19.19. Desde ella podemos tanto ejecutar el servicio como acceder a su descripción pulsando el enlace descripción de servicios. El resultado será un documento XML. (véase figura 19.20).

Figura 19.19. Página obtenida al ejecutar el proyecto

Tal como se aprecia en el código siguiente, la descripción WSDL de nuestro servicio se compone, claramente, de varias secciones distintas.

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:tm=
        "http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soapenc=
        "http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:s0="http://www.fcharte.com"
```

```
targetNamespace="http://www.fcharte.com"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
 <s:schema attributeFormDefault="qualified"
  elementFormDefault="qualified"
  targetNamespace="http://www.fcharte.com">
 <s:element name="Horas">
  <s:complexType />
 </s:element>
 <s:element name="HorasResponse">
  <s:complexType>
   <s:sequence>
    <s:element minOccurs="1" maxOccurs="1"
      name="HorasResult" nillable="true"
      type="s:string" />
   </s:sequence>
  </s:complexType>
 </s:element>
 <s:element name="string" nillable="true"
      type="s:string" />
 </s:schema>
</types>
<message name="HorasSoapIn">
 <part name="parameters" element="s0:Horas" />
</message>
<message name="HorasSoapOut">
 <part name="parameters"
       element="s0:HorasResponse" />
</message>
<message name="HorasHttpGetIn" />
<message name="HorasHttpGetOut">
 <part name="Body" element="s0:string" />
</message>
<message name="HorasHttpPostIn" />
<message name="HorasHttpPostOut">
 <part name="Body" element="s0:string" />
</message>
<portType name="ServicelSoap">
 <operation name="Horas">
  <input message="s0:HorasSoapIn" />
  <output message="s0:HorasSoapOut" />
 </operation>
</portType>
<portType name="ServicelHttpGet">
 <operation name="Horas">
  <input message="s0:HorasHttpGetIn" />
  <output message="s0:HorasHttpGetOut" />
 </operation>
</portType>
<portType name="ServicelHttpPost">
 <operation name="Horas">
  <input message="s0:HorasHttpPostIn" />
  <output message="s0:HorasHttpPostOut" />
 </operation>
</portType>
<binding name="ServicelSoap" type="s0:ServicelSoap">
 <soap:binding
   transport="http://schemas.xmlsoap.org/soap/http"
```

```
    style="document" />
  <operation name="Horas">
   <soap:operation
     soapAction="http://www.fcharte.com/Horas"
     style="document" />
   <input>
    <soap:body use="literal" />
   </input>
   <output>
    <soap:body use="literal" />
   </output>
  </operation>
 </binding>
<binding name="Service1HttpGet"
          type="s0:Service1HttpGet">
  <http:binding verb="GET" />
  <operation name="Horas">
   <http:operation location="/Horas" />
   <input>
    <http:urlEncoded />
   </input>
   <output>
    <mime:mimeXml part="Body" />
   </output>
  </operation>
 </binding>
<binding name="Service1HttpPost"
          type="s0:Service1HttpPost">
  <http:binding verb="POST" />
  <operation name="Horas">
   <http:operation location="/Horas" />
   <input>
    <mime:content
        type="application/x-www-form-urlencoded" />
   </input>
   <output>
    <mime:mimeXml part="Body" />
   </output>
  </operation>
 </binding>
 <service name="Service1">
  <port name="Service1Soap"
        binding="s0:Service1Soap">
   <soap:address
   location="http://localhost/ ServicioHora.asmx" />
  </port>
  <port name="Service1HttpGet"
        binding="s0:Service1HttpGet">
   <http:address
   location="http://localhost/ ServicioHora.asmx" />
  </port>
  <port name="Service1HttpPost"
        binding="s0:Service1HttpPost">
   <http:address
   location="http://localhost/ ServicioHora.asmx" />
  </port>
 </service>
</definitions>
```

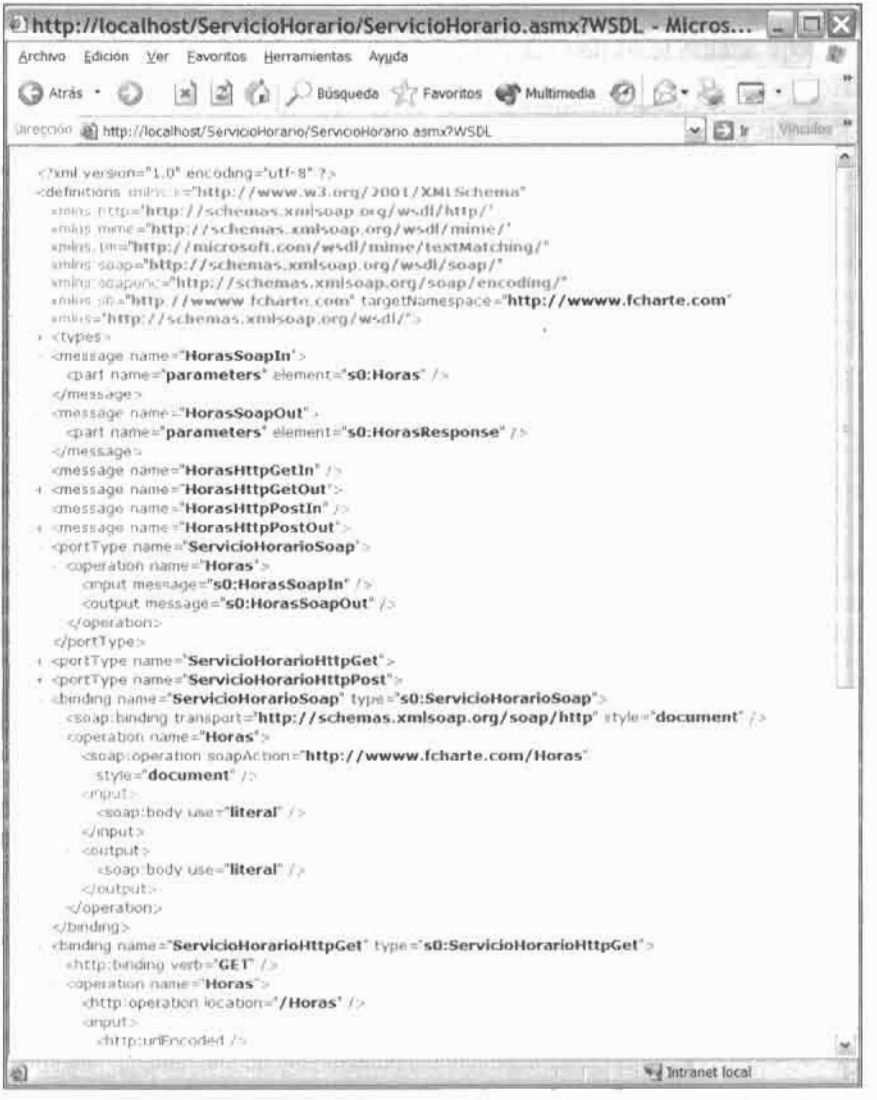


Figura 19.20. Descripción WSDL tal y como aparece en Internet Explorer

Todo el documento se encuentra contenido en un árbol cuyo elemento raíz es la etiqueta `<definitions>`, en cuyo interior podemos distinguir varias partes. En la primera se definen los tipos de datos utilizados en el servicio, en este caso `HorasResponse` que se define como una cadena de caracteres que puede ser nula.

A continuación encontramos la enumeración de los mensajes que pueden enviarse al servicio y los puertos de comunicación. Estos describen el enlace necesario para poder utilizar el servicio mediante SOAP, una solicitud `GET` de http o un `POST` de ese mismo protocolo. Finalmente, encontramos una descripción del servicio y las direcciones y puertos de acceso.

Consumo de un servicio Web

Ya tenemos un servicio web creado y, tras compilarlo, instalado en nuestro sistema, pero no sabremos si realmente funciona o no hasta que lo consumamos. Tal y como se ha visto en la descripción WSDL, podríamos utilizar con tal fin el protocolo HTTP, bien a través del método `GET` o con `POST`, o el protocolo SOAP.

Si pretendemos acceder al servidor desde un cliente Web, como es Internet Explorer, lo normal es que nos sirvamos de HTTP. Realmente no necesitamos diseñar una página ni nada parecido, puesto que basta con introducir la solicitud indicada en la propia descripción WSDL. Inténtelo con el siguiente URI. `http://localhost/ServicioHorario/ServicioHorario.asmx/Horas`, asumiendo que ha utilizado los mismos nombres del ejemplo desarrollado previamente. Debe aparecer una respuesta como la de la figura 19.21. Es un documento XML conteniendo el resultado de la ejecución del servicio.

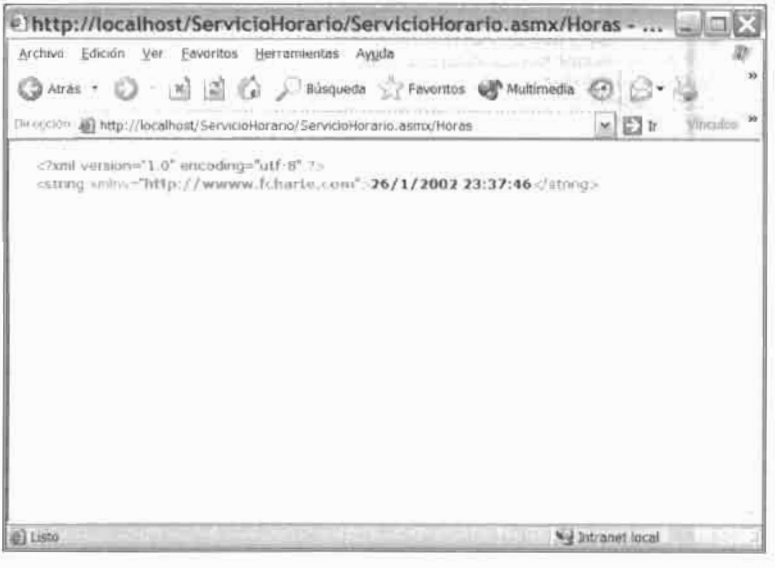


Figura 19.21. Consumimos el servicio introduciendo una solicitud HTTP

Haciendo en el URI referencia directa al servicio Web, sin indicar nombre de método alguno, conseguiremos que ASP.NET obtenga de la WSDL toda la información necesaria para crear una página dinámica en la que se indiquen todos los métodos que existen, los parámetros que toman, sus valores de retorno, etc. En esa página, además, existirán los controles necesarios para que podamos facilitar parámetros y efectuar llamadas. Es una forma rápida de comprobar el funcionamiento de un servicio.

Introduzca el URL `http://localhost/ServicioHorario/ServicioHorario.asmx` y observe lo que ocurre. En el documento devuelto haga clic sobre el enlace **Horas** para acceder a la página, similar a la de la figura 19.22, en

la que puede ver toda la información existente sobre el método. Como se vio antes, también puede abrir la descripción WSDL usando el enlace apropiado.

Figura 19.22. Página generada dinámicamente por ASP.NET para comprobar el funcionamiento de un servicio Web

Aunque es éste un método cómodo y rápido para comprobar el funcionamiento de los servicios Web, desde luego no será el preferido para que los clientes lo utilicen. Lo habitual será crear una aplicación que, sirviendo de interfaz para el usuario final, sea la que consuma el servicio. En los puntos siguientes, por tanto, lo que haremos será actuar como consumidores de ese servicio desarrollando aplicaciones para el usuario.

Un servicio Web puede ser consumido desde cualquier tipo de aplicación, ya se ejecute en Windows nativamente, sea una aplicación Web o utilice otro sistema operativo, sin importar el lenguaje que se haya utilizado. Para poder consumir el servicio que acabamos de escribir, asumiendo que la aplicación consumidora estará en cualquier punto y no precisamente en el ordenador donde se ejecuta el servicio, necesitaremos lo que se conoce habitualmente como un *proxy*.

El *proxy* no es más que un módulo de código que, usado en el cliente o consumidor, hace aparecer como local un servicio que se ejecuta remotamente, de tal forma que su consumo resulta igual de simple que si utilizásemos un componente cualquiera. Para generar este *proxy* tan sólo tenemos que usar una opción de Visual Basic .NET.

Asumiendo que hemos iniciado un nuevo proyecto, por ejemplo una aplicación Windows desde la que pretendemos consumir el servicio, utilizaremos el menú emergente de la carpeta Referencias, en el Explorador de soluciones, para añadir una nueva referencia Web, como se ha hecho en la figura 19.23. Esta opción también está disponible para cualquier otro tipo de proyecto, no es exclusiva de las aplicaciones basadas en formularios Windows.

Figura 19.23. Añadimos al proyecto una referencia externa

De inmediato nos encontraremos con la ventana mostrada en la figura 19.24. En ella tenemos distintas opciones para localizar el servicio Web que deseamos consumir: usar uno de los registros UDDI de Microsoft, buscar servicios locales o, finalmente, introducir en la parte superior el URL del servicio.

En nuestro caso, suponiendo que estamos desarrollando la aplicación en el mismo ordenador donde se encuentra el servicio, pulsaríamos el último enlace de la ventana o, alternativamente, introduciríamos el URL en la barra Dirección. Veríamos aparecer la página en la que se describe el servicio, desde la cual podemos acceder también al módulo WSDL. Terminaríamos pulsando el

botón **Agregar referencia** que hay en la parte inferior (véase figura 19.25) para importar la referencia en nuestro proyecto. En ese momento Visual Basic .NET se encargará de crear el *proxy* para poder consumir ese servicio.

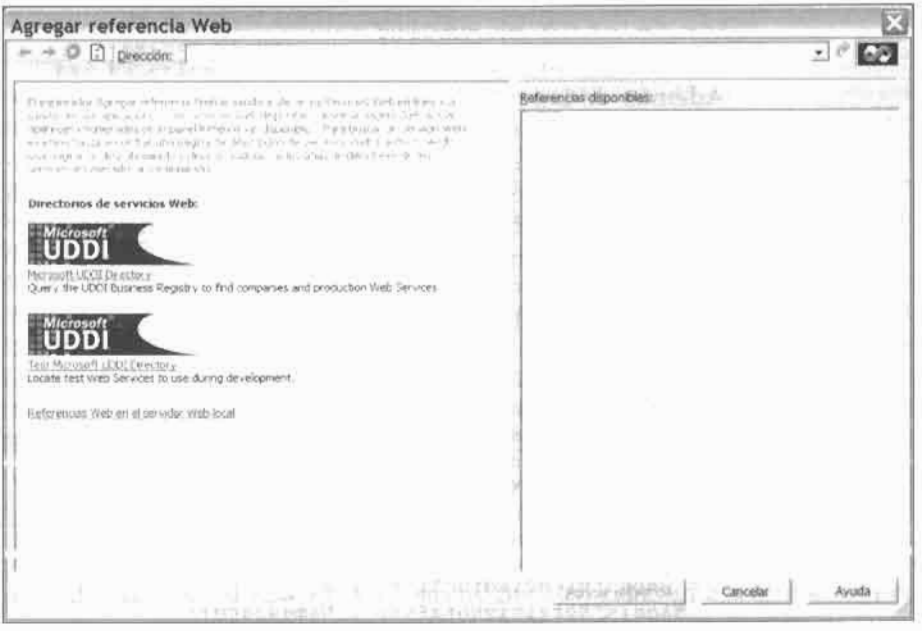


Figura 19.24. Tenemos diferentes opciones para localizar el servicio

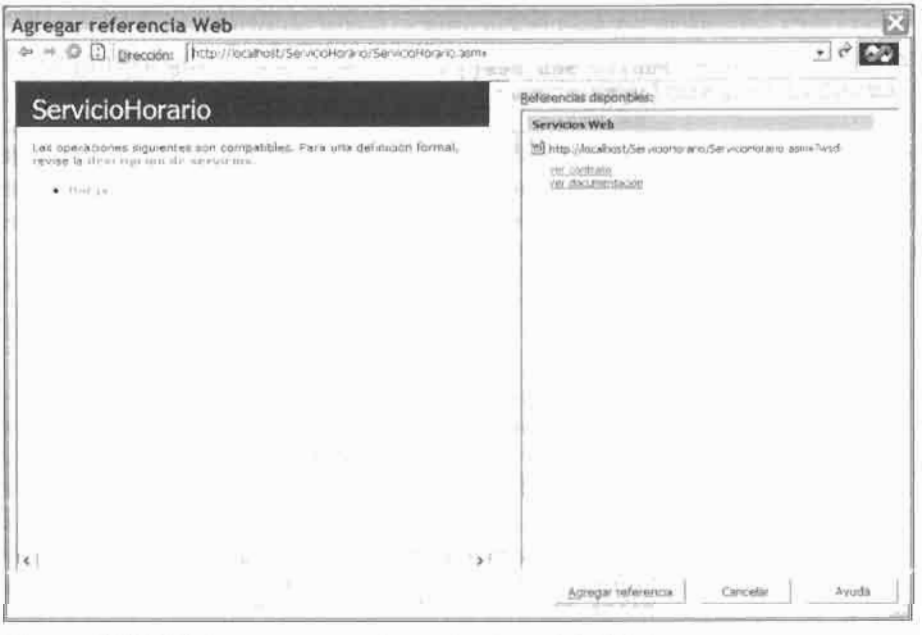


Figura 19.25. Obtenemos una referencia al servicio Web

Importada la referencia verá cómo se añade al Explorador de soluciones un elemento llamado localhost que contiene varios módulos, entre ellos la descripción WSDL del servicio Web.

¿Dónde esta el *proxy*?

Además de los tres módulos que inicialmente podemos ver en la carpeta localhost, el proceso de importación de la referencia al servicio Web ha generado otro que no aparece en el Explorador de soluciones. Puede verlo accediendo, por ejemplo desde el Explorador de archivos de Windows, a la carpeta `Referencias web\localhost` del proyecto. Ese módulo adicional, llamado en este caso `ServicioHorario.vb`, es el que contiene el código del *proxy* propiamente dicho.

Si abre este archivo encontrará un código similar al siguiente:

```
Imports System
Imports System.Diagnostics
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Xml.Serialization

Namespace localhost

  < WebServiceBindingAttribute( _
   Name:="ServicioHoraSoap", Namespace]:= _
   "http://www.fcharte.com")> _
  Public Class ServicioHorario
  Inherits SoapHttpClientProtocol

    < DebuggerStepThroughAttribute()> _
    Public Sub New()
      MyBase.New
      Me.Url = _
       "http://localhost/ServicioHorario/ServicioHorario.asmx"
    End Sub

    < DebuggerStepThroughAttribute(), _
     SoapDocumentMethodAttribute( _
     "http://www.fcharte.com/Horas", _
     RequestNamespace:="http://www.fcharte.com", _
     ResponseNamespace:="http://www.fcharte.com", _
     Use:= Description.SoapBindingUse.Literal, _
     ParameterStyle:= SoapParameterStyle.Wrapped)> _
     Public Function Horas() As String
       Dim results() As Object = _
         Me.Invoke("Horas", New Object(-1) {})
       Return CType(results(0),String)
     End Function
    < DebuggerStepThroughAttribute()> _
     Public Function BeginHoras( _
     ByVal callback As System.AsyncCallback, _
     ByVal asyncState As Object) As IAsyncResult
        Return Me.BeginInvoke("Horas", _
          New Object(-1) {}, callback, asyncState)
```

```
        End Function
      < DebuggerStepThroughAttribute()>  _
        Public Function EndHoras(  _
        ByVal asyncResult As IAsyncResult) As String
          Dim results() As Object =  _
              Me.EndInvoke(asyncResult)
              Return CType(results(0),String)
        End Function
      End Class
  End Namespace
```

Observe que la clase generada en el *proxy* está derivada de la clase `SoapHttpClientProtocol`, utilizando el protocolo SOAP para comunicarse con el servidor que ofrece el servicio.

Asumiendo que podemos obtener la descripción WSDL de cualquier servicio desde un directorio universal, como el perseguido por UDDI, estaríamos en disposición de poder consumirlo directamente generando este *proxy* y dando los pasos que se describirán en el punto siguiente.

La clase cuenta con un método, `Horas()`, que es el encargado de llamar al método homónimo del servicio remoto.

Uso del servicio Web desde el consumidor

Para poder acceder al servicio Web desde nuestro programa tendremos, por tanto, que crear un objeto de la clase generada en el *proxy*, en este caso `ServicioHorario`.

Dicha clase se encuentra en el ámbito con nombre `Consumidor.localhost`, mientras que el formulario de nuestra aplicación está en el ámbito `Consumidor`. Eso significa que tenemos las dos opciones siguientes: importar el ámbito `Consumidor.localhost` o utilizar una referencia tipo `localhost.ServicioHora`.

Insertamos en el formulario Windows, que actualmente está vacío, un control `Label` y un botón. Hacemos doble clic sobre éste e introducimos el código siguiente:

```
Dim MiServicio As localhost.ServicioHorario = _
   New localhost.ServicioHorario()
Label1.Text = MiServicio.Horas()
```

Es todo lo que necesitamos. Cada vez que se pulse el botón se solicitará la hora al servicio Web y se mostrará en la etiqueta de texto. En este caso tenemos el servicio y el consumidor en el mismo ordenador pero, en la práctica, el servicio podría encontrarse en cualquier servidor del mundo. En la figura 19.26 puede ver el programa en funcionamiento.

Podemos hacer el servicio Web mucho más complejo, añadiendo nuevos métodos, y el cliente en lugar de una aplicación Windows puede ser una aplicación Web tipo ASP.NET, un componente u otro servicio. El esquema de desarrollo, no obstante, siempre será el descrito en los puntos previos.

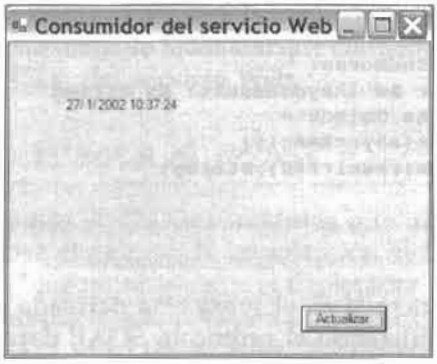


Figura 19.26. El consumidor en funcionamiento

Puntos clave

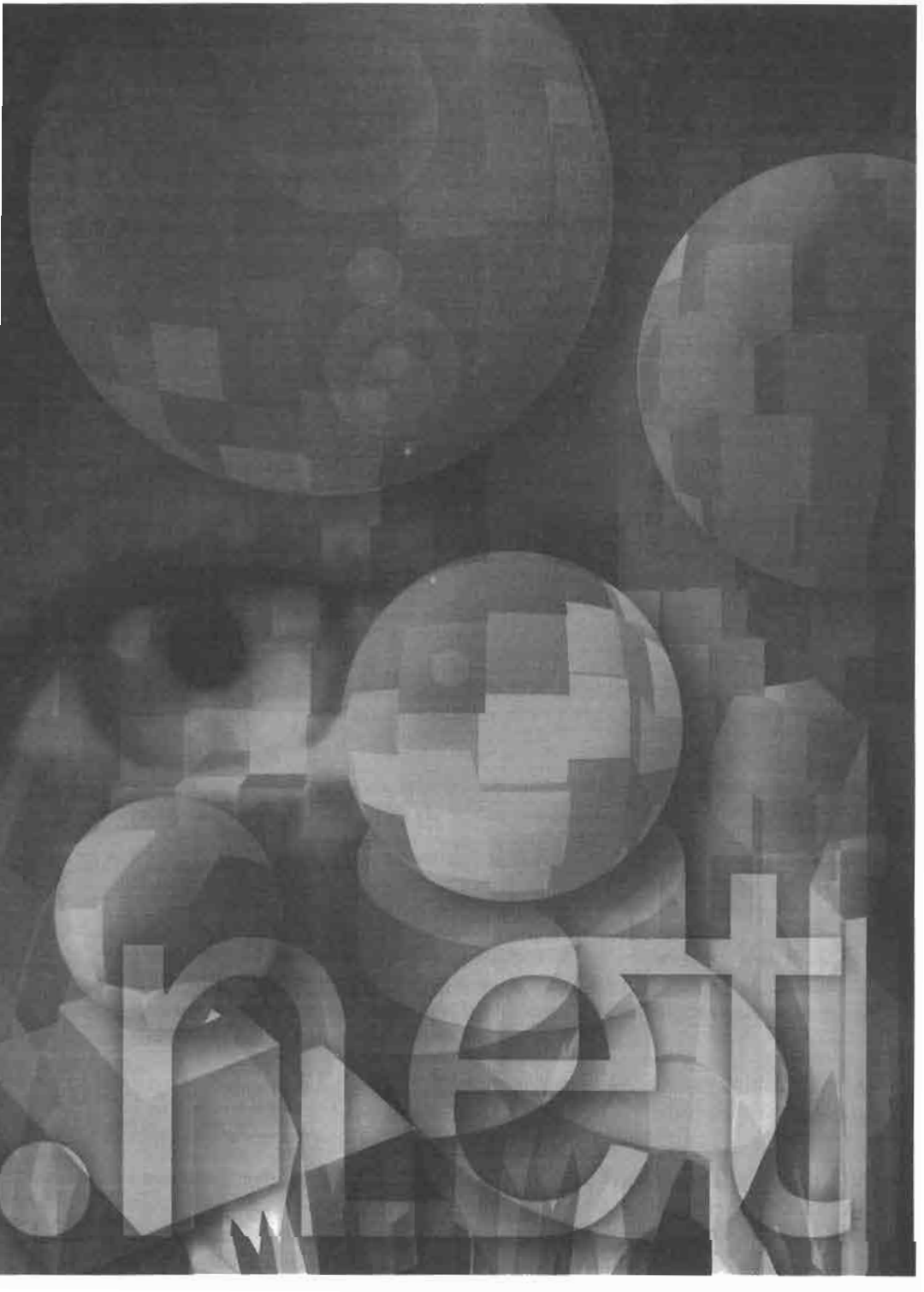
- Un servicio Web es un componente que se ejecuta en un servidor y al que es posible acceder desde cualquier cliente, sin importar el sistema operativo, hardware o lenguaje utilizado para desarrollarlo.
- Partiendo del estándar XML, se han desarrollado diversos protocolos y lenguajes a propósito para facilitar el desarrollo y la comunicación entre servicios Web.
- Mediante el lenguaje WSDL se describe la estructura de un servicio Web: funciones que pone a disposición del cliente, canales para comunicarse con él.
- UDDI es un servicio de directorio que tiene por objetivo simplificar la publicación y localización de servicios Web.
- Para comunicar a los consumidores con un servicio se utiliza generalmente el protocolo SOAP, definido como un protocolo de llamadas remotas que usa HTTP como transporte.
- En torno a XML han surgido muchos otros estándares del W3C, como XSL, XSD, XSLT y XPath que tienen utilidad no sólo en el campo de los servicios Web, sino también en otros muy distintos como la gestión de bases de datos.
- Visual Basic .NET dispone de editores y diseñadores que reconocen la sintaxis de XML y XSD, haciendo más fácil la creación de esquemas y edición de datos contenidos en documentos XML.
- Gracias al asistente de creación de servicios Web de Visual Basic .NET, el único trabajo manual del que tenemos que hacernos cargo es la definición de las funciones del servicio, marcándolas como accesibles desde el exterior mediante el atributo `<WebMethod>`.

- El punto de entrada a un servicio Web es un módulo `asmx` que será procesado por el servidor, de igual forma que el punto de entrada de una aplicación Web es un módulo `aspx`.
- ASP.NET se encarga de generar automáticamente el módulo WSDL de descripción del servicio a partir del código Visual Basic .NET que lo compone.
- Simplemente importando una referencia a un servicio web, utilizando la correspondiente opción del Explorador de soluciones, Visual Basic .NET genera automáticamente el *proxy* necesario para poder utilizarlo.

Resumen

Como ha podido ver en este capítulo, la Web puede utilizarse para algo más que la publicación de documentos más o menos dinámicos. Gracias a recursos como SOAP, XML, HTTP y WSDL es posible el ofrecimiento de servicios universales, que pueden ser consumidos independientemente de plataformas hardware y software, de sistemas operativos y lenguajes de programación.

Hemos desarrollado un servicio Web simple y visto los pasos que hemos de dar para consumirlo desde una aplicación basada en formularios Windows, generando por el camino un *proxy* que, desde Visual Basic .NET, queda oculto. El motor que hay detrás de los servicios Web, en el caso de la plataforma .NET, es ASP.NET.


.net

20 Fundamentos de tratamiento de datos

Como apuntábamos en el capítulo dedicado a los servicios de entrada y salida, la mayoría de las aplicaciones actuales, independientemente de su finalidad, siempre tienen una necesidad: almacenar información para posteriormente poder recuperarla, tratarla, modificarla y, en general, manipularla según las necesidades y objetivos que se persigan.

En el capítulo siguiente veremos que Visual Basic .NET, y su entorno, aportan todos los elementos necesarios para tratar bases de datos, ya sean éstas locales, por ejemplo una base de datos de Microsoft Access, o estén alojadas en un servidor, caso en el que existirá un gestor de bases de datos tipo Microsoft SQL Server.

Antes de conocer la forma de tratar bases de datos con Visual Basic, sin embargo, tendremos que adquirir unos conocimientos mínimos: qué es una base de datos, una tabla, una columna o una fila, cómo se crea una base de datos con SQL Server, qué es SQL y para qué sirve serán algunos de los temas que tratemos en este capítulo. Si está acostumbrado a tratar con bases de datos posiblemente no necesite leer este capítulo, pero nunca está de más recordar conceptos.

Dado que Visual Studio .NET incorpora la edición de escritorio de SQL Server 2000, y que es fácil conseguir una versión de prueba de este producto en su edición completa, tanto en este capítulo como en el siguiente asumiremos que

vamos a trabajar con él. La mayoría de los conceptos son generales a todas las bases de datos y sistemas RDBMS, pero las opciones y características concretas de SQL Server 2000 son, lógicamente, distintas a las de Access, FoxPro u otro servidor de datos.

¿Qué es una base de datos?

Su propio nombre sugiere que una base de datos es un lugar en el que es posible alojar información, una especie de bolsa o cajón donde podemos depositar datos diversos. Lo cierto es que una base de datos puede contener mucho más que datos, configurándose en realidad en un objeto vital para la correcta gestión de la información que precisan las empresas.

Una base de datos es el objeto lógico que sirve para el almacenamiento y recuperación de los datos desde un dispositivo, generalmente una unidad de disco, pero también debe asegurar la integridad de esos datos basándose en reglas previamente definidas. La base de datos debe autodescribirse, de tal forma que la información que contiene pueda ser correctamente interpretada.

Al trabajar con una base de datos es fundamental el rendimiento a la hora de recuperar datos, por ejemplo el resultado de la ejecución de una consulta. Uno de los mecanismos para acelerar estas acciones es el uso de índices, otra entidad que también estaría contenida en el interior de la base de datos. Asimismo, en la base de datos pueden existir vistas prefabricadas de la información.

La propia lógica de funcionamiento puede estar almacenada en la base de datos, de manera que las operaciones más usuales se definen como procedimientos almacenados que, posteriormente, pueden ejecutarse cuando sea necesario. Acciones como la creación de una nueva cuenta de cliente, la totalización de un pedido o la búsqueda de los productos pertenecientes a una cierta familia pueden codificarse como procedimientos almacenados. De esta forma la base de datos no sólo contiene los datos, sino también los procesos necesarios para manipularlos.

En el caso particular de las bases de datos de SQL Server 2000, éstas también almacenan funciones de usuario y permisos que limitan o describen los privilegios que tienen los usuarios sobre la información.

¿Qué es un RDBMS?

Las aplicaciones actuales nunca manipulan directamente la información contenida en una base de datos, sino que usan un intermediario, conocido como RDBMS (*Relational Database Management System*), al que envían instrucciones sobre lo que debe hacer. El RDBMS traduce esas instrucciones en acciones concretas: creación de una tabla, extracción de todas las filas que cumplan un cierto criterio, eliminación de un dato y los relacionados, etc. También se conoce a los RDBMS como *servidores de datos*.

SQL Server 2000 es un RDBMS, mientras que los archivos que manipula son las bases de datos, de manera análoga a como IIS (*Internet Information Server*) es un servidor web y las páginas ASP.NET son los datos sobre los que trabaja. La existencia de un RDBMS permite centralizar no sólo la información, sino también la lógica necesaria para mantener su coherencia.

Cuando las necesidades de una aplicación, en cuanto a almacenamiento y gestión de datos, son muy simples, puede recurrirse a una base de datos local o de escritorio, como las de Access, en lugar de recurrir a un RDBMS.

Estructura física de una base de datos

Desde el punto de vista físico, una base de datos está compuesta de uno o varios archivos en disco, en el caso de SQL Server 2000 son dos como mínimo. Aunque es posible usar directamente espacio de disco sin particionar ni formatear para almacenar bases de datos, lo usual es usar una partición ya existente, con formato FAT, FAT32 o NTFS, para almacenar los archivos de la base de datos.

Como se ha dicho, los archivos necesarios son al menos dos si usamos SQL Server: un archivo principal con datos y un archivo de registro de transacciones. De manera adicional pueden utilizarse archivos de datos secundarios, por ejemplo si se prevé que la base de datos va a tener un tamaño considerable. También es útil el uso de varios archivos cuando éstos se alojan en varias unidades de discos distintas, ya que potencialmente el rendimiento es superior al trabajar dichas unidades en paralelo.

Los archivos de datos, tanto el principal como los secundarios, contendrán los datos propiamente dichos, así como el resto de información descrita en el punto anterior: índices, procedimientos almacenados, vistas, etc.

El archivo de registro de transacciones, por su parte, se utiliza para conservar la base de datos en un estado íntegro, facilitando su recuperación en caso de que fuese necesaria. Para ello en él se almacena un registro de todos los cambios llevados a cabo sobre la base de datos. En caso necesario también es posible contar con varios archivos de registro.

Nombres de archivo

Los nombres de las bases de datos se forman, generalmente, a partir de una raíz facilitada por el operador que la crea y unas terminaciones y extensiones fijas. Dicho operador, no obstante, puede establecer el nombre que prefiera, seleccionando asimismo el destino en el que se alojarán los archivos. Suponiendo que estuviésemos creando una base de datos llamada `Librería`, los nombres típicos para el archivo principal de datos y para el archivo de registro serían `Librería_Data.MDF` y `Librería_Log.LDF`, respectivamente.

En caso de que vayan a existir múltiples archivos, ya sean de datos o de registro, los nombres de los archivos secundarios también pueden ser establecidos

por el creador, siendo la extensión habitual NDF. Cada uno de estos archivos, como se ha dicho antes, podría alojarse en una carpeta e incluso unidad distintas. Es buena idea, por ejemplo, mantener los archivos de datos en una unidad distinta a la que contiene los archivos de registro de transacciones. Aparte de mejorar el rendimiento, el fallo de una unidad no conllevará la pérdida de ambos elementos.

La figura 20.1 es una representación gráfica de la estructura física que tendría una base de datos con un archivo principal de datos y un archivo de registro de transacciones. Desde el punto de vista lógico, el que veríamos desde el Administrador corporativo de SQL Server, el entorno de Visual Basic .NET o una aplicación, la base de datos sería una entidad muy distinta a dos simples archivos en disco.

Figura 20.1. Una base de datos lógica contenida en dos archivos físicos en disco

Estructura lógica de una base de datos

Para poder usar una base de datos no es necesario conocer con detalle su estructura física, de igual forma que para conducir un automóvil tampoco necesitamos saber cómo funciona el motor. Sí precisaremos, por el contrario, conocer la lógica del propio automóvil: pedales, cambio de velocidad, volante, etc. De forma análoga, para trabajar con una base de datos lo que sí debemos conocer es su estructura lógica, ya que será ésta la que debamos utilizar continuamente.

En ese archivo principal de datos que mencionábamos en el punto anterior Librería_Data.MDF por poner un ejemplo, pueden existir tablas de datos, vistas de esas tablas, índices que aceleren el trabajo con ellas, procedimientos almacenados con la lógica para operar sobre ellas, etc. Puede utilizar el Administrador corporativo de SQL Server y seleccionar una base de datos de las facilitadas como ejemplo, pubs es una de ellas, para ver cuál es su contenido.

En los apartados siguientes se introducen, de manera breve, algunos de los elementos que podemos encontrar en una base de datos con el fin de que al mencionarlos sepamos de qué estamos hablando. En la figura 20.2 puede ver una imagen del Administrador corporativo mostrando las categorías de elementos existentes en la base de datos pubs.

Figura 20.2. Desde el Administrador corporativo podemos ver los elementos de una base de datos

Tablas

Con este nombre es conocida la entidad básica de almacenamiento de información en una base de datos. Todos los datos que podamos añadir, ya sea mediante una herramienta estándar o bien a través de una aplicación a medida, se almacenarán en tablas. Al efectuar una consulta o al abrir una vista, los datos se recuperan asimismo de una o más tablas.

Una tabla está compuesta de varias filas de datos, también conocidos como registros, cada una de las cuales cuenta con el mismo conjunto de columnas o campos. Cada una de estas columnas tiene un nombre y un tipo que identifica la clase de información que puede contener. Las columnas se disponen horizontalmente, de izquierda a derecha, mientras que las filas lo hacen verticalmente, de arriba a abajo, dando lugar a la tabla de datos.

Generalmente una base de datos contiene múltiples tablas, existiendo enlaces entre ellas que permiten establecer relaciones uno a uno, uno a muchos, etc. Un ejemplo simple sería una base de datos con información de facturación. En una tabla tendríamos el encabezado de cada pedido existente, mientras que

otra almacenaría las líneas de cada pedido. En este caso concreto, el enlace entre ambas tablas permitiría crear una relación uno a muchos, obteniendo a partir de la cabecera de un pedido todas las filas que forman parte de él.

Índices

Al trabajar con tablas de tamaño considerable, con muchos miles o, incluso, millones de filas, el proceso de recuperación de datos puede verse afectado de manera considerable. Es lógico si tenemos en cuenta que para, por ejemplo, encontrar todas las filas de un pedido, sería preciso recorrer la tabla de líneas de pedido completa, de principio a fin.

Un mecanismo común para acelerar esa operación son los índices. Un índice es una lista de claves con una estructura tal que el servidor puede realizar búsquedas de forma muy rápida en ella. Las claves pueden estar formadas por el contenido de una o varias columnas de una tabla. Los índices se almacenan, al igual que las tablas, en los archivos de datos mencionados anteriormente.

Además de acelerar la búsqueda de información, un índice también puede ser usado para establecer el orden en que se almacenarán las filas en una tabla.

Cada vez que se inserta, modifica o elimina una tabla de la cual depende uno o varios índices, el servidor tiene que actualizar no sólo la tabla de datos sino también todos los índices que existan a fin de que sean consistentes con la información actual.

Es fácil deducir que cuantos más índices existan más tiempo será necesario para efectuar cualquiera de esas operaciones.

Vistas

Al trabajar con una base de datos es habitual que, de forma reiterada, se soliciten más o menos los mismos conjuntos de datos, para lo que se compondrán sentencias de consulta SQL prácticamente idénticas. Con el fin de evitar este trabajo repetitivo se idearon las vistas, que podríamos definir como consultas ya almacenadas en la base de datos.

Una vista puede ser abierta y tratada como si fuese una tabla pero, en realidad, no contiene datos sino que los extrae de una o más tablas a partir de una sentencia de consulta previamente definida. Es posible incluso utilizar la vista para actualizar la información devuelta, aunque con algunas limitaciones.

Mediante las vistas es posible automatizar tareas como la creación de enlaces entre tablas para obtener datos de varias de ellas. Una vista podría, por ejemplo, facilitarnos todos los datos de un pedido conjuntamente con sus líneas facilitando tan sólo el código de ese pedido.

Procedimientos almacenados

El servidor SQL Server no sólo es capaz de manipular información, gestionando tablas e índices, procesando consultas, etc., sino que, además, también

es capaz de ejecutar procedimientos escritos en un lenguaje llamado Transact-SQL en el caso de SQL Server. Estos procedimientos se denominan procedimientos almacenados, dado que están almacenados en la propia base de datos.

Gracias a los procedimientos almacenados, un administrador puede crear no sólo la estructura de las bases de datos con que trabajarán las aplicaciones, sino que también puede implementar la lógica para la manipulación de dichos datos.

Continuando con el ejemplo que ya ha sido citado en puntos anteriores, al crear una base de datos para almacenar información sobre pedidos, con tablas de pedidos, líneas, clientes, etc., también podríamos definir un procedimiento almacenado que se encargase de crear un nuevo pedido, de devolver el total de un pedido, etc.

Los procedimientos almacenados se ejecutan mucho más rápido que las sentencias SQL que puedan enviarse desde una aplicación, dado que se almacenan en la base de datos ya compilados. De esta forma, SQL Server no tiene más que ejecutar el código, sin necesidad de efectuar un análisis sintáctico y una compilación previa.

Nota

Existen procedimientos almacenados que, en lugar de ejecutarse al ser invocados de manera explícita, se ejecutan automáticamente en el momento en que se desencadena un cierto suceso o evento. Estos procedimientos son conocidos habitualmente como *triggers*, disparadores o desencadenadores.

Otros elementos lógicos

En los cuatro puntos anteriores se han abordado los elementos lógicos de una base de datos que podríamos considerar de primer nivel: tablas, índices, vistas y procedimientos almacenados. Éstos, no obstante, no son los únicos que podemos encontrar. Una base de datos también puede alojar diagramas de datos, reglas y valores predeterminados, así como tipos de datos y funciones definidas por el usuario.

Dos de las categorías de objetos importantes, desde el punto de vista de la seguridad, son los usuarios y las funciones. Inicialmente el único usuario existente en una base de datos nueva de SQL Server es el conocido como dbo, iniciales de *database owner*. Este usuario representa al dueño de la base de datos que, en principio, es el usuario que la ha creado.

Las funciones representan conjuntos de permisos que pueden asignarse o negarse a los usuarios.

Existe un grupo predeterminado de funciones y también podemos crear las nuestras propias. Las funciones ya existentes, llamadas estándar, se identifican con tareas como la lectura de datos, la administración de seguridad o la realización de copias de seguridad.

Introducción a SQL

Microsoft SQL Server es un sistema de bases de datos relacional que, como todos los demás, utilizan el lenguaje SQL para facilitar al usuario la ejecución de sentencias tanto de manipulación como de selección de datos. Conocer este lenguaje es, por tanto, básico para trabajar no ya con SQL Server, sino con cualquier otro RDBMS y, en general, para poder desarrollar aplicaciones con tratamiento de bases de datos en Visual Basic .NET.

SQL es un lenguaje no procedural y orientado a datos. Esto significa que no tiene estructuras de control como las de los lenguajes de programación, ya que su finalidad es devolver conjuntos de datos, modificarlos y, en el más amplio sentido, manipular datos. Aunque SQL es un lenguaje estándar, ampliamente aceptado, existen algunas diferencias puntuales entre el dialecto de SQL que entiende cada RDBMS. Las construcciones básicas, no obstante, son suficientemente genéricas como para funcionar en prácticamente cualquier sistema.

En los puntos siguientes no se tratará de explicar todos los detalles del lenguaje SQL, sino más bien transmitir los conceptos necesarios para que, en el capítulo siguiente, pueda desenvolverse sin problemas al tratar los comandos de bases de datos de ADO.NET.

El Analizador de consultas SQL

Para poder hacer pruebas con el lenguaje SQL necesitamos saber dónde podemos introducir nuestras sentencias SQL para poder ejecutarlas y que el servidor efectúe las operaciones que le indiquemos. En este caso la herramienta que necesitamos es el **Analizador de consultas SQL** de SQL Server.

Para abrir el **Analizador de consultas SQL** deberá desplegar el menú **Herramientas** del **Administrador corporativo**, seleccionando la opción que puede verse en la figura 20.3. No confunda ésta con la opción **Analizador SQL** que hay debajo y que no tiene nada que ver aunque el nombre sea similar. Se trata de una herramienta diferente cuya finalidad es examinar el flujo SQL mientras se trabaja con una aplicación.

Aunque las posibilidades de esta herramienta son muchas, para el fin que nos ocupa tan sólo tiene que saber cómo introducir las sentencias SQL y cómo ejecutarlas. Como se aprecia en la figura 20.4, correspondiente a la ventana del **Analizador de consultas SQL**, la mayor parte del área de trabajo está ocupada por una ventana en la que podemos introducir texto. Será aquí, por tanto, donde tecleemos las sentencias a ejecutar. Cuando procesemos las órdenes, pulsando el botón que puede ver destacado encima o mediante la tecla **F5**, aparecerá un panel en la parte inferior con mensajes y resultados.

En la parte izquierda de la ventana hay una página que puede usar para explorar los diversos objetos que puede utilizar. De esta forma no tiene por qué memorizar los nombres de tablas, columnas, etc. Incluso puede tomar cualquiera de esos objetos y arrastrarlo hasta la ventana de texto para insertar el identificador correspondiente.

Figura 20.3. Opción para abrir el Analizador de consultas SQL

Figura 20.4. Aspecto del Analizador de consultas SQL

Componentes y derivados de SQL

El lenguaje SQL (*Structured Query Language*, Lenguaje estructurado de consulta) cuenta con múltiples construcciones cuya finalidad, tal como se ha dicho

antes, es manipular datos en algún sentido. Aprender SQL no es tan difícil como aprender un nuevo lenguaje de programación, aunque los derivados de SQL por cada fabricante llegan a niveles similares.

Las distintas sentencias de SQL pueden englobarse en dos grupos, que podríamos decir que son los componentes de SQL. Éstos son DDL (*Data Definition Language*, Lenguaje de definición de datos) y DML (*Data Manipulation Language*, Lenguaje de manipulación de datos). Tanto DDL como DML son partes integrales de SQL y forman parte del lenguaje estándar.

Después encontramos los derivados de SQL o dialectos basados en dicho lenguaje. En el caso concreto de SQL Server, Microsoft se ha basado en SQL para crear un derivado llamado Transact-SQL. Éste cuenta con los elementos necesarios para, por ejemplo, poder crear procedimientos almacenados.

DDL

Muchos usuarios piensan que el lenguaje SQL se utiliza sólo y exclusivamente para efectuar consultas en una base de datos, de tal manera que conociendo la cláusula `SELECT` y poco más ya hay suficiente. Hay otra parte del lenguaje SQL, sin embargo, cuya finalidad es definir estructuras de datos. Es lo que se denomina Lenguaje de definición de datos o DDL.

Mediante este subconjunto de SQL podemos crear bases de datos, definir tablas, crear índices, vistas y procedimientos almacenados. También puede ser usado para efectuar modificaciones sobre todas esas entidades, por ejemplo añadiendo columnas a una tabla o cambiando la intercalación de una base de datos. Las diversas operaciones que podemos efectuar mediante DDL son distintas construcciones basadas principalmente en tres órdenes distintas: `CREATE`, `ALTER` y `DROP`. Más adelante, en este mismo capítulo, tendrá ocasión de conocerlas. Con ellas podremos crear las tablas que necesitemos, estableciendo sus claves y relaciones.

DML

El subconjunto de SQL dedicado a la manipulación de los datos, denominado DML, es sin duda el más conocido. Su finalidad no es actuar sobre las estructuras de la base de datos, sino sobre la información que hay contenida en ellas. Con DML podríamos obtener todo el contenido de una tabla, combinar columnas relacionadas de varias tablas, actualizar datos, añadirlos o eliminarlos.

Las cuatro órdenes más importantes de DML son `SELECT`, `INSERT`, `DELETE` y `UPDATE`, cada una de las cuales, como podrá ver a continuación, puede contar con múltiples cláusulas y variantes. En ocasiones podemos encontrarnos con combinaciones de DML y DDL. Es posible, por ejemplo, utilizar una consulta para crear una nueva tabla. Estas construcciones en ocasiones tienen una sintaxis dependiente del RDBMS que esté utilizándose.

Transact-SQL

Como se indicaba antes, cada sistema RDBMS suele usar un dialecto o derivado de SQL con características específicas que, en cierta manera, le permiten

aprovechar las singularidades de ese servidor en particular. En el caso de Microsoft SQL Server 2000, dicho dialecto recibe el nombre de Transact-SQL.

Aunque todo lo que va a ver en el resto de este capítulo forma parte del lenguaje Transact-SQL, puesto que éste es el derivado de SQL que entiende SQL Server, realmente nos centraremos en las construcciones más estándar, prácticamente idénticas en todos los RDBMS actuales y en herramientas como Microsoft Access. Sí tendríamos que usar Transact-SQL, por ejemplo, a la hora crear procedimientos almacenados.

Definición de datos

Para poder manipular datos primero es necesario definir su estructura. Por ello comenzaremos conociendo algo de DDL, el lenguaje para definición de datos. Con él podremos crear una base de datos, tablas en su interior, efectuar modificaciones e incluso eliminar objetos. Puede ir comprobando las explicaciones dadas ejecutando los comandos con el Analizador de consultas SQL, viendo los resultados en esa misma ventana o en las distintas ramas mostradas en el Administrador corporativo.

Con DDL podremos crear elementos con la orden `CREATE`, modificarlos con la orden `ALTER` y eliminarlos mediante la orden `DROP`. Cada una de éstas tiene variantes para operar sobre bases de datos, tablas, vistas, procedimientos almacenados, etc.

Cualquiera de estas tres órdenes irá seguida de uno de los identificadores enumerados en la tabla 20.1. Con ellos se indica la entidad que va a crearse, modificarse o eliminarse.

Tabla 20.1. Identificación de entidades a manipular

Identificador	Entidad
`DATABASE`	Base de datos
`TABLE`	Tabla
`VIEW`	Vista
`PROCEDURE`	Procedimiento almacenado
`TRIGGER`	Desencadenador

Además de los enumerados en esta tabla, existen otros objetos que tan sólo son aplicables con alguna de las órdenes. `CREATE`, por ejemplo, puede utilizarse para crear un índice, pero éste no puede ser modificado con `ALTER`.

Creación de una base de datos

La primera entidad que deberíamos crear es la propia base de datos, para lo cual usaremos la sentencia `CREATE DATABASE`. El único parámetro imprescindible es el nombre lógico de la base de datos, aunque opcionalmente pueden facilitarse muchos más.

Utilizando el Analizador de consultas SQL, la ejecución de la simple sentencia `CREATE DATABASE Pruebas` crearía una base de datos llamada `Pruebas` utilizando parámetros por defecto para los nombres de los archivos físicos, su tamaño, crecimiento y resto de propiedades. En la figura 20.5 puede ver el resultado obtenido tras ejecutar la sentencia. Puede utilizar el Administrador corporativo para comprobar que la base de datos ha sido creada.

Figura 20.5. Podemos crear una nueva base de datos con una sentencia muy simple

En el momento de creación de la base de datos podemos especificar tanto el nombre del archivo de datos como el del archivo de registro de transacciones, pudiendo también establecer su tamaño inicial, tamaño máximo y el factor de crecimiento. Los parámetros para el archivo de datos se facilitarán, entre paréntesis, tras la cláusula `ON`, mientras que los del archivo de registro, que son equivalentes, lo harán detrás de las palabras `LOG ON`.

```
CREATE DATABASE nombre
ON (parámetros)
LOG ON (parámetros)
```

Los parámetros posibles, en ambos casos, son `NAME`, `FILENAME`, `SIZE`, `MAXSIZE` y `FILEGROWTH`, facilitando el nombre lógico del archivo, su camino y nombre físico en disco, el tamaño inicial, el tamaño máximo y el crecimiento. La mayoría de estos parámetros son opcionales.

Observe la imagen mostrada en la figura 20.6. La primera sentencia que aparece, `DROP DATABASE Pruebas`, elimina la base de datos del servidor puesto que, de lo contrario, no podríamos volver a crearla. Debajo la creamos de nue-

vo estableciendo algunos de los parámetros del archivo de datos y de registro de transacciones.

Figura 20.6. Al crear una base de datos podemos establecer los atributos de los archivos de datos y registro de transacciones

De forma similar, usando parámetros opcionales, podríamos establecer el orden de intercalación por defecto para la base de datos o definir un grupo de archivos de almacenamiento.

Creación de tablas

Otra de las variantes de la orden `CREATE` es `CREATE TABLE` que, como es obvio, usaremos para definir tablas en el interior de una base de datos. El primer paso será seleccionar la base de datos en la que va a efectuarse la operación, para lo cual usaremos la sentencia `USE` **`nombre_base`**. A continuación ya podremos crear las tablas que necesitemos.

El parámetro más importante, y único imprescindible a la hora de crear una tabla aparte de su nombre, es la lista de columnas que deberá facilitarse entre paréntesis. Por cada columna debemos facilitar un nombre y un tipo. De forma adicional podríamos especificar si será la clave principal, cuenta con restricciones, hace referencia a una clave de otra tabla, etc. La sintaxis general, obviando todas esas posibilidades, es la siguiente:

```
CREATE TABLE nombre
(
  nombre_columna tipo,
  nombre_columna tipo ...
)
```

Lógicamente, para crear una tabla deberíamos conocer los tipos de datos que hay disponibles. Dos de los más habituales son `smallint` y `varchar`. El primero de ellos es un entero mientras que el segundo es una secuencia de caracteres, debiendo facilitarse entre paréntesis su longitud máxima.

Suponga que deseamos definir, en la base de datos recién creada en el punto previo, una tabla para contener datos de libros. En la figura 20.7 puede ver el comando usado. En la ventana de mensajes tan sólo se indica que el comando se ha completado con éxito, lo cual significa que la tabla ha sido creada. Una vez más, puede comprobar la operación desde el Administrador corporativo.

Figura 20.7. Seleccionamos la base de datos y creamos una tabla en ella

Utilizando opciones como `PRIMARY KEY`, `CHECK` y `DEFAULT` podríamos indicar qué columna actúa como clave principal, definir restricciones de datos o facilitar valores por defecto. Son algunas de las posibilidades existentes.

Modificación de una tabla

Con una correcta planificación y un estudiado diseño las modificaciones a la estructura de una base de datos deberían ser, teóricamente, mínimas. Hay que tener en cuenta que este tipo de operaciones podrían, en determinados casos, provocar pérdidas de información, algo que no es admisible para una base de datos que esté utilizándose en un entorno de explotación.

Para efectuar cambios en una tabla usaríamos la sentencia `ALTER TABLE` seguida del nombre de la tabla y las opciones que correspondan. Básicamente podemos efectuar tres operaciones distintas: añadir columnas, modificarlas o eliminarlas. También puede utilizarse esta construcción para modificar, elimi-

nar o añadir opciones de una columna, como pueden ser los valores por defecto o las restricciones.

Si deseamos cambiar una determinada columna, por ejemplo para modificar su longitud o incluso el tipo de dato que le habíamos asignado en un principio, debemos usar la sintaxis siguiente:

```
ALTER TABLE nombre_tabla
  ALTER COLUMN nombre_columna nuevo_tipo
```

En caso de que el cambio sea una reducción de longitud o la modificación del tipo podría provocar, como acaba de decirse, una pérdida de información. Al ejecutar una sentencia de este tipo, como se ha hecho en la figura 20.8, en el panel de resultados el Analizador de consultas SQL nos indicará cuántas filas de datos han resultado afectadas. En este caso, puesto que la tabla está aún vacía, los cambios en la estructura no afectan a los datos.

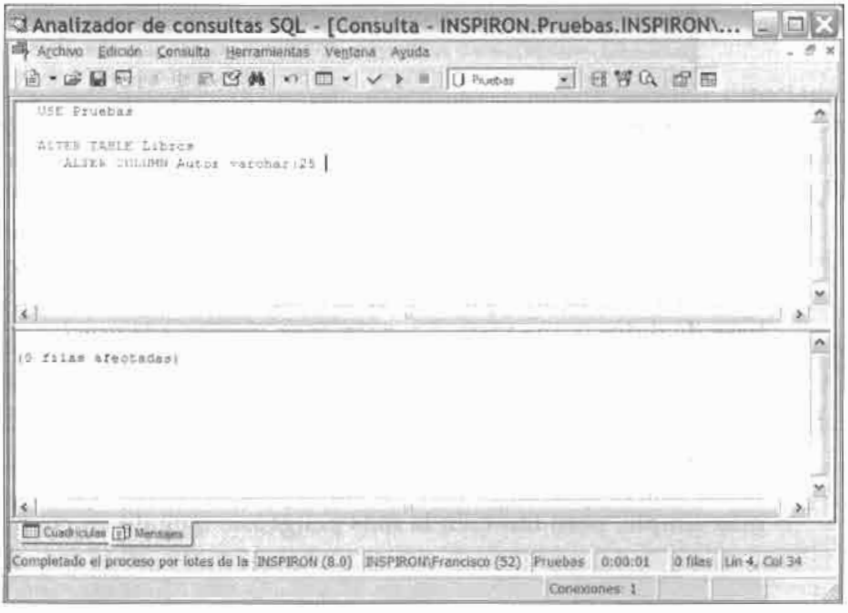


Figura 20.8. Reducimos la longitud de una de las columnas de la tabla `Libros`

Las construcciones para añadir y eliminar columnas de la tabla son similares, como puede verse a continuación. En la figura 20.9 puede ver cómo se ha añadido una columna para el ISBN y se ha eliminado la columna que contendría el nombre del autor.

```
ALTER TABLE nombre_tabla
  ADD nombre_columna tipo

ALTER TABLE nombre_tabla
  DROP COLUMN nombre_columna
```

Figura 20.9. Añadimos una nueva columna y eliminamos una de las existentes

Otras operaciones

En los puntos anteriores hemos creado y modificado entidades como una base de datos, una tabla y sus columnas. Tanto `CREATE` como `ALTER` pueden utilizarse, asimismo, para crear un índice, modificar la configuración de una base de datos o una vista. La sintaxis es similar a la que hemos conocido en pasos previos. La tercera orden del lenguaje DDL que habíamos mencionado, `DROP`, la hemos usado en uno de los ejemplos para eliminar la base de datos y así poder volver a crearla. Si en lugar de la palabra `DATABASE` ponemos detrás de `DROP` los parámetros `TABLE`, `INDEX`, `VIEW`, etc., seguidos del nombre de la tabla, índice o vista, estaremos eliminando el objeto referenciado. Es la orden más simple, pero también la más peligrosa, téngalo en cuenta.

Nota

Efectúe todas las pruebas que desee, con las distintas órdenes DDL que ha conocido, siempre sobre una base de datos de pruebas. No utilice las entregadas como ejemplo con SQL Server 2000, como `pubs`, y mucho menos las bases de datos de sistema que, como `master`, son imprescindibles para un correcto funcionamiento del servidor.

Manipulación de datos

Como se indicaba anteriormente, las operaciones fundamentales de manipulación de datos son cuatro: selección, inserción, modificación y eliminación.

Estas cuatro operaciones se corresponden con las órdenes SELECT, INSERT, UPDATE y DELETE, respectivamente. Cada una de estas órdenes tiene una estructura básica, relativamente simple, que puede complementarse con diversas opciones.

A la hora de seleccionar datos, modificarlos o eliminarlos es posible, utilizando algunas de esas opciones, limitar la acción a un cierto conjunto de datos. Veamos primero cómo podemos efectuar las tareas generales y, después, entraremos en el estudio de algunas de esas opciones.

Inserción de nuevas filas

Comenzaremos por la inserción de datos, ya que sin datos en las tablas no es posible efectuar operaciones de selección, actualización ni borrados. La orden a utilizar, por tanto, es INSERT, cuya sintaxis básica podemos ver a continuación.

```
INSERT INTO nombre_tabla
  VALUES (valor, valor ...)
```

La palabra INTO es opcional, podemos obviarla y facilitar directamente el nombre de la tabla detrás de la orden INSERT. Los valores irán entre paréntesis y separados por comas, delimitándose entre comillas simples los que sean cadenas de caracteres. Asumiendo que tenemos la tabla Libros de la base de datos Pruebas con las últimas modificaciones efectuadas, podríamos utilizar las sentencias que se ven en la figura 20.10 para añadir un par de filas con sendos libros.

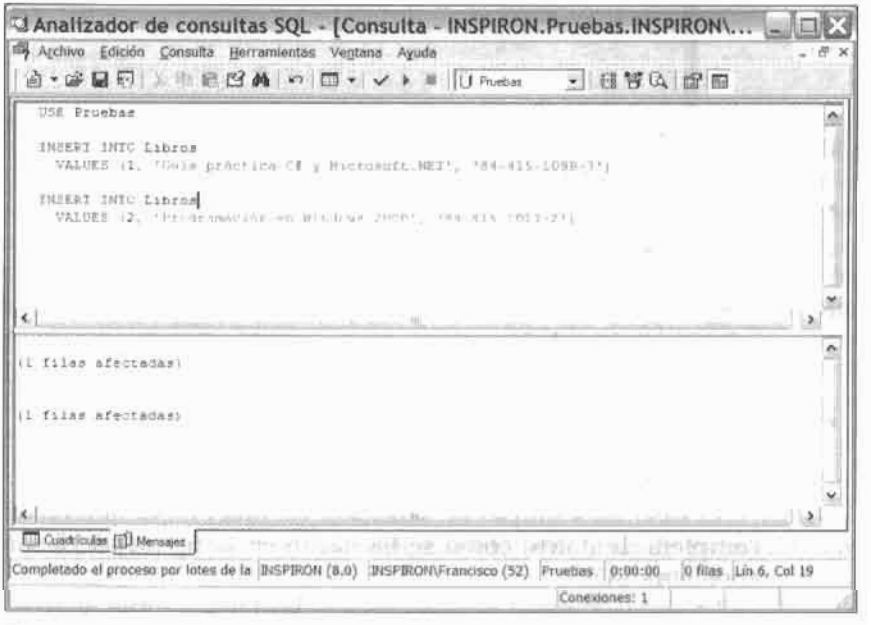


Figura 20.10. Insertamos dos filas de datos en la tabla

En este ejemplo se facilitan tantos valores como columnas existen en la tabla, respetando también el orden. Si no se entregan valores para todas las columnas, o el orden es distinto, entonces la sintaxis a utilizar sería la siguiente:

```
INSERT INTO nombre_tabla (nombre_columna, ...)
  VALUES (valor, ...)
```

La figura 20.11 presenta dos ejemplos en los cuales se utiliza esta sintaxis. En la primera inserción se facilitan los tres datos, pero en un orden diferente al que ocupan las columnas en la tabla. En el segundo caso se entregan valores tan sólo para dos de las tres columnas, por lo que la tercera tomará el valor NULL. Esto es posible porque al definir la tabla no hemos indicado que esta tabla no aceptase valores nulos.

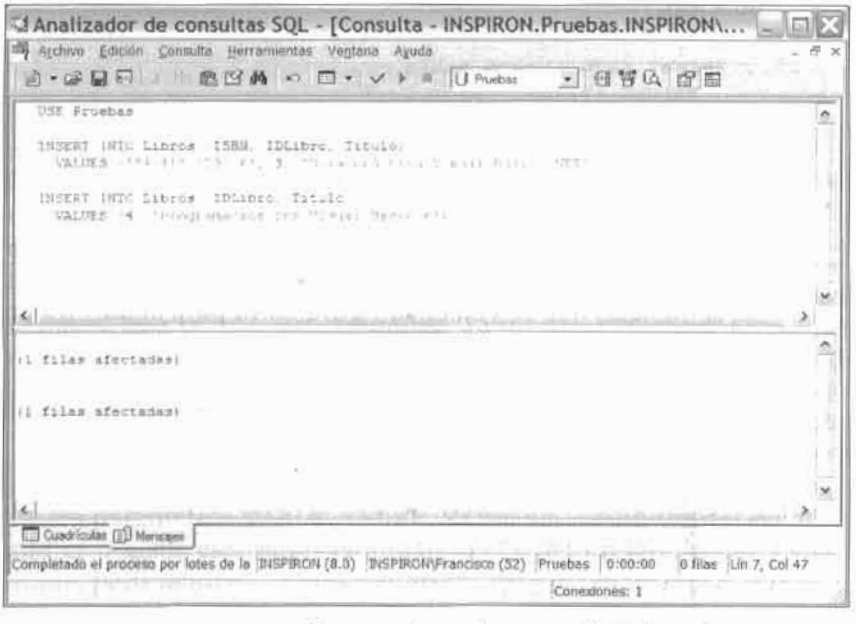


Figura 20.11. Insertamos filas con los valores en distinto orden o con menos valores

Selección de datos

La orden más conocida del lenguaje DML, y en general de SQL, es sin duda alguna SELECT. Ésta se utiliza para seleccionar un conjunto de datos determinado, obteniendo una serie de filas y columnas que pueden ser una tabla completa, una parte de una tabla o datos de múltiples tablas relacionadas entre sí.

El caso más simple de selección es aquel en el que recuperamos una tabla completa de datos, como se ha hecho en la figura 20.12 con el Analizador de consultas SQL.

La sintaxis, como puede verse, es bastante simple: detrás de SELECT disponemos un asterisco y tras la palabra FROM el nombre de la tabla.

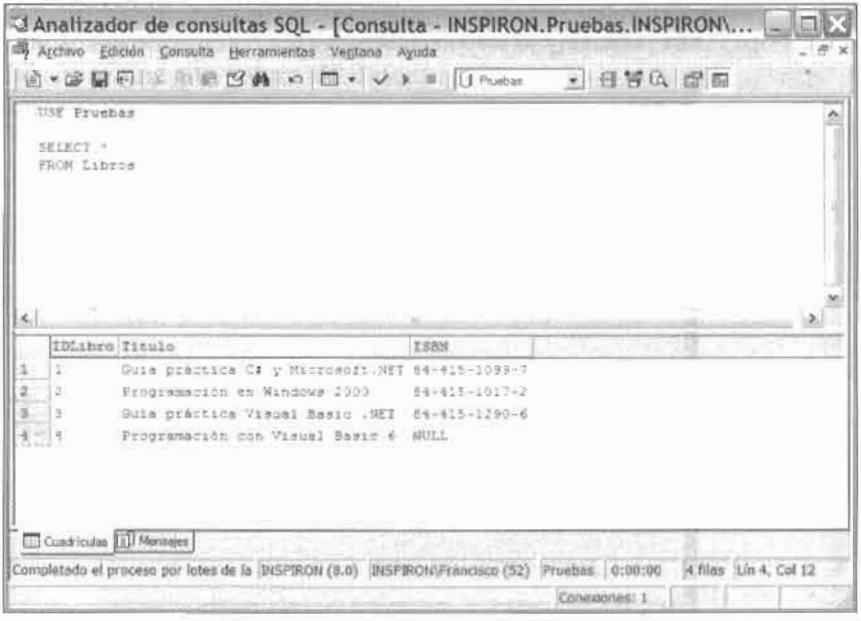


Figura 20.12. Seleccionamos todas las filas y columnas de la tabla `Libros`

Podemos limitar las columnas a recuperar sustituyendo el asterisco por los nombres de las columnas que nos interesen, separando unas de otras mediante comas. Con la sentencia siguiente, por ejemplo, obtendriamos todos los titulos e ISBNs de la tabla de libros, pero no sus códigos.

```
SELECT Titulo, ISBN
FROM Libros
```

Si lo que deseamos limitar son las filas devueltas, a la construcción anterior tendremos que añadir un nuevo apartado formado por la palabra `WHERE` y una expresión de selección. En dicha operación pueden utilizarse diversos operadores para buscar cierto dato, todos aquellos que sean menores, mayores, distintos, similares, etc.

En la figura 20.13 puede ver cómo se ha utilizado el apartado `WHERE`, conjuntamente con el operador `LIKE`, para encontrar todos los libros cuyo título se inicia con la palabra `Programación`. En la parte inferior puede ver los resultados devueltos.

Modificación de datos

La actualización de datos es una tarea habitual, consistente en modificar el valor de una o más columnas pertenecientes a una o más filas. Lo habitual es limitar la acción a un determinado conjunto de filas, usando para ello la misma clausula `WHERE` que hemos usado anteriormente para seleccionar datos.

En este caso la orden que nos interesa es `UPDATE` y la sintaxis básica es la mostrada a continuación:

```
UPDATE nombre_tabla
  SET nombre_columna = nuevo_valor
```

Figura 20.13. Seleccionamos algunas de las filas de la tabla `Libros`

Para facilitar el nuevo valor podemos usar un dato constante, como los que previamente hemos insertado en las filas de la tabla `Libros`, o una expresión. Podríamos, por ejemplo, sumar un determinado número de ejemplares a una hipotética tabla `Almacen` que ya tuviésemos creada.

Con la sintaxis anterior la actualización afectaría a todas las filas del conjunto de datos, sin excepción. Suponga que desea corregir el título de uno de los libros que ha introducido previamente, para lo cual pretende utilizar una sentencia como la siguiente:

```
UPDATE Libros
  SET Titulo='Programación DNA en Windows 2000'
```

¿Qué conseguiría con esto? Perder los títulos de todos los libros que hay en la tabla y establecer el mismo para todos ellos. Para conseguir lo que quiere tiene que limitar la operación a la fila concreta sobre la que debe efectuarse el cambio, usando la cláusula `WHERE` y una expresión que identifique dicha fila de manera única. Podría, por ejemplo, comprobar el actual título o basarse en el ISBN, como se ha hecho en el ejemplo de la figura 20.14. A continuación puede ejecutar una consulta para verificar los cambios.

Eliminación de datos

La última operación que podemos efectuar, de las cuatro mencionadas con anterioridad, es el borrado de filas de una columna. A diferencia de lo que ocu

rre con la selección, actualización e inserción, casos en los que puede afectarse sólo a algunas columnas de la fila sobre la que se trabaja, el borrado tiene lugar siempre sobre una fila entera de datos.

Figura 20.14. Cambiamos el título de uno de los libros

Podemos usar la orden DELETE tanto para borrar todo el contenido de una tabla, simplemente facilitando el nombre de ésta detrás, como para eliminar un determinado conjunto de filas. En este segundo caso deberemos seleccionar dicho conjunto usando la cláusula WHERE, como se ha hecho previamente con la actualización.

La sentencia para eliminar un libro concreto, por ejemplo facilitando su ISBN, podría ser la siguiente:

```
DELETE FROM Libros
  WHERE ISBN='84-415-1017-2'
```

Puede comprobar de inmediato, con una sentencia SELECT, el efecto causado sobre la tabla. Con ésta ya conoce las cuatro operaciones fundamentales, por lo que puede insertar datos, seleccionarlos, modificarlos y eliminarlos. En principio, es todo lo que necesita para trabajar con el Analizador de consultas SQL y hacer pruebas.

Relaciones entre tablas

En este momento tenemos en nuestra base de datos de pruebas una sola tabla, por lo que difícilmente podemos relacionarla con ninguna otra. Para ver cómo podemos usar la orden SELECT para obtener datos de dos tablas tendremos

primero que crear otra, a continuación insertar datos en ella y, por último, ejecutar la consulta.

Supongamos que ahora queremos crear la tabla `Almacen` para contener el número de ejemplares que hay de cada título. Esta tabla tendría tan sólo dos columnas: `IDLibro` y `Cantidad`. La crearíamos con la siguiente sentencia SQL:

```
CREATE TABLE Almacen
(
   IDLibro smallint,
   Cantidad smallint
)
```

En principio, como es lógico, la tabla estará vacía. Tendríamos que insertar algunas filas facilitando por cada una de ellas un código de libro y la cantidad que tenemos de él. Podemos utilizar cualquier valor e insertar, por ejemplo, tres filas.

```
INSERT INTO Almacen VALUES (1, 3)
INSERT INTO Almacen VALUES (2, 5)
INSERT INTO Almacen VALUES (5, 1)
```

Ahora tenemos una tabla con el código, título e ISBN de varios libros, y otra que contiene una serie de códigos y número de ejemplares. Si obtenemos el contenido de la tabla `Almacen`, estos datos no nos dirán mucho por sí solos a menos que conozcamos de memoria los códigos de los libros. Es mucho más fácil combinar ambas tablas creando la oportuna relación que, en este caso, se basará en el código del libro. La sentencia sería la siguiente:

```
SELECT *
FROM Libros, Almacen
WHERE Libros.IdLibro = Almacen.IdLibro
```

Observe la expresión que sigue a la cláusula `WHERE`. En ella solicitamos aquellas filas de la combinación de tablas en que la columna `IdLibro` de la primera sea igual a la columna del mismo nombre en la segunda. En este caso ambas columnas tienen el mismo nombre, pero esto no es absolutamente necesario. El resultado, como puede ver en la figura 20.15, es que junto al título de cada libro ahora también vemos el número de ejemplares disponibles.

Si en lugar de `*` hubiésemos especificado el nombre de algunas columnas, tras la orden `SELECT`, el resultado podría ser más claro. Pruebe, por ejemplo, a solicitar tan sólo el título de cada libro y la cantidad disponible.

Puntos clave

- Una base de datos es un recipiente lógico donde almacenar información, tanto datos como la estructura de esos datos y la lógica necesaria para tratarlos.

Figura 20.15. Relacionamos las dos tablas para obtener datos conjuntos

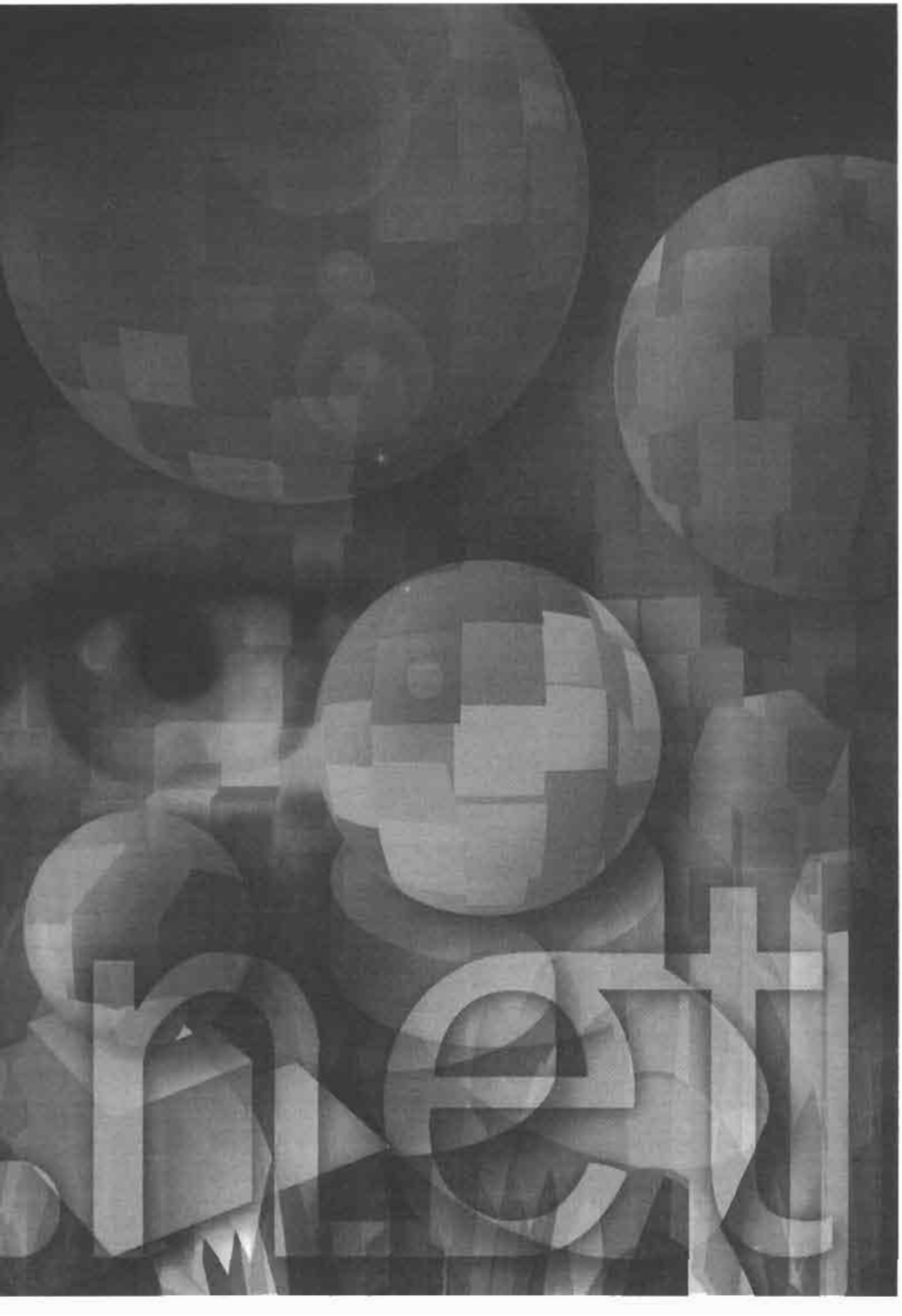
- Físicamente una base de datos está compuesta de uno o más archivos en disco.
- En la práctica nunca es necesario abrir esos archivos y trabajar directamente con ellos, porque las aplicaciones utilizan los RDBMS como intermediarios para la gestión de datos.
- Los elementos básicos de una base de datos son las tablas, conjuntos de filas de datos compuestas cada una de ellas de una serie de columnas.
- El acceso a los datos puede acelerarse definiendo los índices adecuados sobre las columnas de las tablas.
- Una base de datos también puede contener vistas, procedimientos almacenados y otros elementos de utilidad.
- Para trabajar con sistemas RDBMS es fundamental conocer el lenguaje SQL. Éste se compone de dos subconjuntos: DDL y DML, utilizándose el primero para manipular las estructuras de la base de datos y el segundo los datos propiamente dichos.

Resumen

En caso de que no conociese previamente SQL Server y el lenguaje SQL, seguramente este capítulo le habrá sido de utilidad para adquirir los conocimientos más básicos y poder realizar diversas operaciones sobre una base de datos

experimental. Estas pruebas le servirán para, posteriormente, saber cómo actuar sobre bases de datos reales.

Si, por el contrario, ya conocía estos conceptos, o incluso es un experto en SQL, lo cierto es que este capítulo le habrá sabido a muy poco. Todas las órdenes que se han explicado cuentan con múltiples opciones y variantes, aparte de las ya explicadas. Aquí es, no obstante, donde le será de utilidad la referencia de Transact-SQL que forma parte de los libros en pantalla de SQL Server.



21
Acceso a datos con ADO.NET

Una necesidad invariable de los desarrolladores de aplicaciones, independientemente de cuál sea la finalidad de éstas, es el acceso a datos, en su más amplio sentido, y el tratamiento de bases de datos, para ser más concretos. En la actualidad dicha necesidad implica, básicamente, el trabajo con dos tipos de fuentes de datos: documentos XML y sistemas RDBMS como SQL Server, Oracle o DB2.

Las soluciones para simplificar el acceso a datos desde los lenguajes de programación más usuales han sido muchas y diversas, generalmente dependientes de un cierto fabricante y, en muchas ocasiones, ligadas a una determinada plataforma, sistema operativo o incluso lenguaje. No obstante, existen algunos hitos importantes que, con el tiempo, han sido fundamentales, como el lenguaje SQL (*Structured Query Language*) o bien el estándar ODBC (*Open Database Connectivity*).

Originalmente Windows no incorporaba mecanismo alguno para facilitar el acceso a datos, lo que causó que diversos fabricantes aportasen productos propios junto a sus herramientas de desarrollo. Así, mientras que con Visual Basic o Visual C++ podía utilizarse DAO (*Data Access Objects*) otros entornos muy conocidos, como Borland Delphi, incorporaban un mecanismo alternativo llamado BDE (*Borland Database Engine*).

La aparición de Windows 2000 marcó un antes y un después. A partir de ese momento era el propio sistema operativo el que incorporaba los mecanismos necesarios para acceder a bases de datos. La base eran los controladores OLE DB para distintos sistemas RDBMS y ADO (*ActiveX Data Objects*) como interfaz de alto nivel para comunicarse con esos controladores. Todas las herramientas

de desarrollo pueden utilizar esas interfaces, a través de COM, y ya no es precisa la redistribución del motor de acceso a datos siempre que el sistema destinatario sea Windows 2000.

Como no podía ser de otra manera, la llegada de la plataforma .NET trae consigo una evolución de los mecanismos de acceso a datos. ADO dejará paso a ADO.NET, un modelo de objetos más avanzado que simplifica la construcción de aplicaciones en múltiples niveles al facilitar el trabajo sobre conjuntos de datos sin necesidad de mantener una conexión continua con el sistema RDBMS. Nuestro objetivo, en este capítulo, es conocer básicamente ADO.NET y la forma de utilizarlo desde Visual Basic .NET.

Modelo de objetos de ADO.NET

ADO.NET es un modelo simplificado de acceso a datos en el que existen básicamente dos elementos: el conjunto de datos desconectado, representado por el objeto `System.Data.DataSet`, y unos proveedores que facilitan la ejecución de operaciones contra un sistema RDBMS. Actualmente tenemos a nuestra disposición adaptadores para acceder a dos tipos de proveedores: `SqlDataAdapter` y `OleDbDataAdapter`, definidos en los ámbitos `System.Data.SqlClient` y `System.Data.OleDb`.

Nota

Está en desarrollo un tercer proveedor, en fase de *beta* en el momento de escribir este libro, que permitirá utilizar ODBC (*Open Database Connectivity*) desde ADO.NET sin necesidad de recurrir al puente OLE DB-ODBC.

El esquema de funcionamiento consiste, básicamente, en utilizar uno de los adaptadores para conectar con el RDBMS, recuperando un conjunto de datos y cerrando la conexión. Ese conjunto, que sería un objeto `DataSet`, puede ser transferido entre aplicaciones y puede operarse sobre él prácticamente como si fuese una base de datos. Finalmente, se restablecería la conexión utilizando el citado proveedor y se enviarían al RDBMS las actualizaciones que fuesen pertinentes.

La representación interna de los datos, por parte del objeto `Dataset`, utiliza el lenguaje XML. La información transferida de un punto a otro, por tanto, es un flujo XML independiente del sistema operativo, plataformas y lenguajes. En la figura 21.1 puede ver una representación gráfica de los elementos que intervienen en el acceso a datos usando ADO.NET.

Estructura de un objeto `DataSet`

Un objeto `DataSet` facilita el trabajo con una base de datos sin necesidad de mantener una conexión continua. Lo que obtenemos es una representación

en memoria de esa base de datos, y observe que decimos base de datos y no simplemente una tabla. Previamente, en el modelo ADO, el objeto principal de trabajo era el `Recordset`: la representación lineal de una sola tabla. Un `DataSet`, por el contrario, puede albergar múltiples tablas y, además, las relaciones existentes entre ellas o las reglas de validación, permitiendo así efectuar operaciones complejas sin necesidad de restablecer la conexión con el RDBMS.

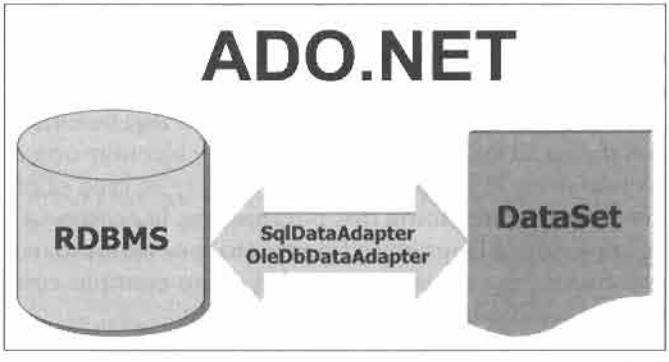


Figura 21.1. Modelo de acceso a datos utilizando ADO.NET

Internamente el `DataSet` mantiene una colección de tablas, estando cada una de ellas representada por un objeto `DataTable`. Cada tabla puede provenir incluso de bases de datos diferentes. Aparte de relaciones y reglas de validación una tabla contendrá, lógicamente, datos. Las distintas filas y columnas se representan mediante objetos `DataRow` y `DataColumn`, respectivamente.

La figura 21.2 es una representación simplificada de la estructura de un `DataSet` que, en este caso concreto, está formado por dos `DataTable`. Cada uno de estos contiene sus propios objetos `DataRow` y `DataColumn`. No aparecen en la figura otros elementos que no contienen datos propiamente dichos, sino información adicional como las citadas relaciones entre tablas o las reglas de validación.

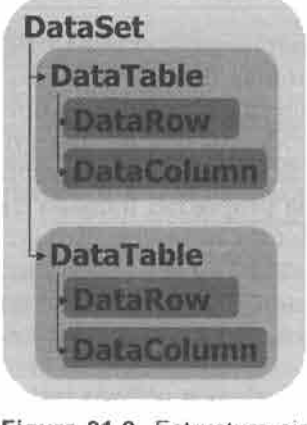


Figura 21.2. Estructura simplificada de un objeto `DataSet`

Creación de un DataSet a partir de una tabla

Para poder utilizar algunas de las propiedades de un objeto DataSet, por ejemplo con el fin de obtener los nombres y columnas de una tabla alojada en una base de datos, primero tendremos que crear el DataSet. Esto es tan fácil como crear cualquier otro objeto de una clase cualquiera. En ese momento tendríamos un DataSet vacío. El siguiente paso sería establecer una conexión con una base de datos y llenar el DataSet, ya sea a partir de una tabla o una consulta que afecte a varias.

Asumiendo que tenemos instalado SQL Server 7, SQL Server 2000 o MSDE (*Microsoft Data Engine*), utilizaríamos un objeto SqlDataAdapter para conectar con una de las bases de datos de ejemplo y ejecutar una consulta. Todo esto puede efectuarse en el mismo momento en que se crea el objeto, gracias a que uno de sus constructores toma dos parámetros: la sentencia SQL a ejecutar y la cadena de conexión. El siguiente fragmento, por ejemplo, conectaría con la base de datos pubs, una de las entregadas como ejemplo con SQL Server, y obtendría todo el contenido de la tabla authors.

```
Dim Comando As SqlDataAdapter = New SqlDataAdapter(_
  "SELECT * FROM authors", "server=INSPIRON;uid=sa;database=pubs")
```

En mi caso concreto se está accediendo a SQL Server 2000, un sistema RDBMS que se ejecuta en el servidor INSPIRON. La cuenta usada es la del administrador de sistema.

De todos los miembros con que cuenta un objeto de la clase SqlDataAdapter en este momento nos interesa tan sólo uno: el método Fill(). Este toma como primer parámetro un objeto DataSet y copia en él el resultado de ejecutar el comando indicado previamente. El segundo parámetro sería el nombre de la tabla creada en el DataSet y, normalmente, coincidirá con la obtenida en la consulta.

Recuperación de la estructura de un DataSet

Dados los dos pasos previos ya tenemos un DataSet con información, concretamente con todas las filas y columnas de una tabla. Podríamos repetir el proceso tantas veces como lo necesitáramos, añadiendo nuevas tablas al mismo DataSet. Veamos ahora cómo podemos obtener la estructura de la información contenida en él, concretamente el nombre de las tablas y las columnas.

El objeto DataSet dispone de una propiedad llamada Tables que es una colección de objetos DataTable. Podemos enumerar dicha colección para recuperar datos de las tablas. Cada DataTable cuenta con propiedades diversas y, entre ellas, una llamada Columns que, como puede imaginar, es una colección de objetos DataColumn. Un objeto de este tipo describe a una columna de la tabla.

Conociendo sólo estos elementos podemos utilizar un ejemplo como el siguiente para generar la salida que puede verse en la figura 21.3. Se trata de una

página ASP.NET que genera una tabla HTML con el nombre y tipo de columna de cada tabla que exista en un DataSet.

```
Dim Conexion As String = "server=localhost;uid=sa;database=pubs"

Dim Comando1 As SqlDataAdapter = New _
   SqlDataAdapter("SELECT * FROM authors", Conexion)

Dim Comando2 As SqlDataAdapter = New _
   SqlDataAdapter("SELECT * FROM titles", Conexion)

Dim MiDataSet As DataSet = New DataSet()
Dim MiTabla As DataTable
Dim MiColumna As DataColumn

Comando1.Fill(MiDataSet, "Autores")
Comando2.Fill(MiDataSet, "Titulos")

For Each MiTabla In MiDataSet.Tables

  Response.Write("Tabla: <b>" & MiTabla.TableName & "</b><hr>" & _
    "<table border=1><tr><td><b>Nombre</b></td>" & _
    "<td><b>Tipo</b></td></tr>")

  For Each MiColumna In MiTabla.Columns
    Response.Write("<tr><td>" & MiColumna.ColumnName & _
      "</td>" & "<td>" & MiColumna.DataType.Name & "</td></tr>")
  Next
  Response.Write("</table><hr>")
Next
```

Nota

Inicie un nuevo proyecto de aplicación Web ASP.NET en lenguaje Visual Basic.NET, añada el ámbito `System.Data.SqlClient` a la lista de sentencias `Imports` y, finalmente, introduzca el código anterior en el método correspondiente al evento `Load` de la página.

Además del nombre y tipo de cada columna, podríamos utilizar las propiedades de la clase `DataColumn` y `DataTable` para mostrar el valor por defecto que tiene cada columna, si permite valores nulos, obtener las reglas de validación de la tabla o bien su clave primaria. Tan sólo hay que consultar la referencia de clases en `System.Data` para encontrar los miembros que necesitamos utilizar.

Figura 21.3. La página mostrando la estructura de las tablas

Control de la conexión

En caso de que pretendamos ejecutar múltiples comandos sobre una misma base de datos, como hemos hecho en el ejemplo del punto previo, en lugar de establecer la conexión, ejecutar y desconectar, empleando algún tiempo en un proceso que después tendremos que repetir, podemos separar lo que es la conexión del comando. Con este fin podemos utilizar el objeto `SqlConnection` para representar la conexión, abriéndola y cerrándola cuando nos convenga.

Nota

Recuerde que todos los objetos del espacio `System.Data.SqlClient` están pensados para trabajar con SQL Server. Si desea usar otro RDBMS utilice el espacio `System.Data.OleDb`, sustituyendo en los nombres de

las clases el prefijo `Sql` por `OleDb`. La conexión con cualquier otra base de datos, por tanto, se efectuaría con un objeto `OleDbConnection`.

Un objeto `SqlConnection` dispone, entre otros, de los métodos `Open()` y `Close()`. Éstos, como podrá suponer, son los encargados de abrir y cerrar la conexión entre el objeto `SqlConnection` y el sistema RDBMS. Mientras está abierto podemos ejecutar todos los comandos que precisemos, sin el retardo que supondría cerrar y volver a abrir la conexión a cada paso.

Nuestro ejemplo anterior podría codificarse sustituyendo la actual declaración de la variable `Conexion` por la siguiente:

```
Dim Conexion As SqlConnection = New _
    SqlConnection("server=INSPIRON;uid=sa;pwd=;database=pubs")
```

Antes de llamar al método `Fill()`, para obtener los datos, utilizaríamos el método `Open()` de `Conexion` para abrir la conexión con la base de datos. Tras recuperar todos los datos usaríamos el método `Close()` para cerrarla. El resultado sería exactamente el mismo, la diferencia es que ahora la conexión y desconexión se efectúa una sola vez, lo cual redundará en un menor tiempo para efectuar las consultas. Esto será especialmente visible si estamos usando una conexión remota.

Acceso a los datos

Hasta ahora hemos estado trabajando con lo que se denomina la *metainformación* de la base de datos, utilizada para describir su estructura. Las tablas de una base de datos contienen, lógicamente, datos útiles. En el caso de la tabla `titles`, por ejemplo, disponemos de columnas con títulos de libros, fechas de publicación, precios, etc.

Los objetos `DataTable` disponen de la propiedad `Rows`, una colección de objetos `DataRow` cada uno de los cuales representa a una fila de datos de la tabla. A su vez, cada fila está compuesta de columnas, tantas como indique la colección `Columns`. Podemos utilizar la propiedad `Rows` como si fuese un arreglo gracias a que cuenta con una propiedad que actúa como indexadora, siendo el índice bien el nombre de la columna o su número de orden.

Efectuando algunos cambios al código de la página ASP.NET usada como ejemplo en los puntos anteriores, podríamos conseguir un resultado como el de la figura 21.4.

Lo que hacemos, básicamente, es recorrer cada fila de la tabla convirtiéndola en una fila de una tabla HTML. Añadiendo algunos controles a la página ASP.NET se puede facilitar, por ejemplo, la ejecución de una consulta sobre la tabla de títulos en lugar de devolverla entera.

```
Dim Conexion As SqlConnection = New SqlConnection( _
   "server=INSPIRON;uid=sa;pwd=;database=pubs")
```

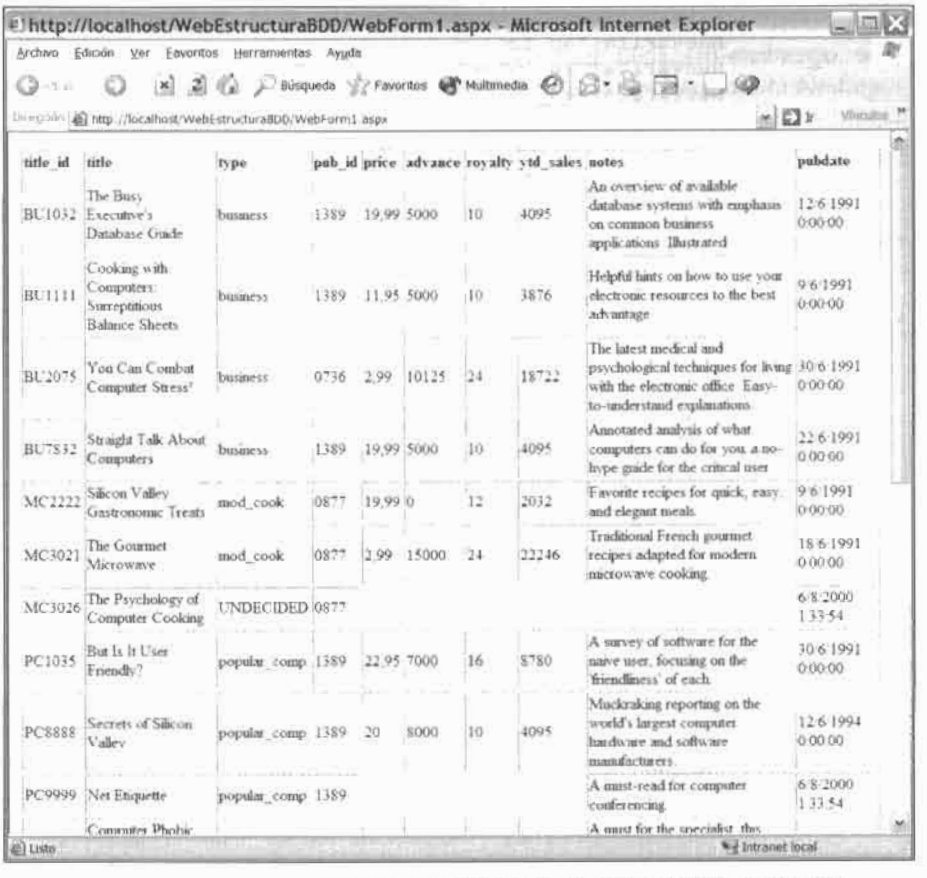
```
Dim Comando As SqlDataAdapter = New _
    SqlDataAdapter("SELECT * FROM Titles", Conexion)

Dim MiDataSet As DataSet = New DataSet()

Comando.Fill(MiDataSet, "Autores")

Dim MiTabla As DataTable = MiDataSet.Tables("Autores")

Dim Columna As DataColumn
Dim Fila As DataRow
Dim Ind As Integer

Response.Write("<table border=1><tr>")

For Each Columna In MiTabla.Columns
  Response.Write("<td><b>" & Columna.ColumnName + "</b></td>")
Next
Response.Write("</tr>")

For Each Fila In MiTabla.Rows
  Response.Write("<tr>")

  For Ind = 0 To MiTabla.Columns.Count - 1
    Response.Write("<td>" & Fila(Ind).ToString() + "</td>")
  Next
  Response.Write("</tr>")
Next
Response.Write("</table>")
```

Las referencias del tipo `Fila(Ind)` o `Fila("title")` pueden usarse no sólo para obtener el valor de un cierto dato, sino también para modificarlo mediante una simple asignación. Hay que tener en cuenta, no obstante, que estamos trabajando con un conjunto de datos sin disponer de una conexión al RDBMS, por lo que los cambios serán visibles, al menos en principio, sólo localmente.

Actualización de los datos

El objeto `SqlDataAdapter` dispone de cuatro propiedades que apuntan a otros tantos objetos `SqlCommand`. Dichas propiedades son `SelectCommand`, `UpdateCommand`, `InsertCommand` y `DeleteCommand`, siendo su finalidad mantener las sentencias y parámetros necesarios para permitir la selección de datos, su actualización, inserción y borrado, respectivamente. Estos comandos se utilizan cuando se efectúa una actualización.

Cualquier cambio que efectuemos en el `DataSet` permanecerá en éste hasta que efectuemos una llamada al método `Update()` del `SqlDataAdapter`. En ese momento se utilizarán los comandos citados para transferir todas las modificaciones al RDBMS.



http://localhost/WebEstructuraBDD/WebForm1.aspx - Microsoft Internet Explorer

Archivo Edición Ver Favoritos Herramientas Ayuda

Búsqueda Favoritos Multimedia

Dirección http://localhost/WebEstructuraBDD/WebForm1.aspx

title_id	title	type	pub_id	price	advance	royalty	ytd_sales	notes	pubdate
BU1032	The Busy Executive's Database Guide	business	1389	19,99	5000	10	4095	An overview of available database systems with emphasis on common business applications. Illustrated.	12/6/1991 0:00:00
BU1111	Cooking with Computers: Surreptitious Balance Sheets	business	1389	11,95	5000	10	3876	Helpful hints on how to use your electronic resources to the best advantage.	9/6/1991 0:00:00
BU2075	You Can Combat Computer Stress!	business	0736	2,99	10125	24	18722	The latest medical and psychological techniques for living with the electronic office. Easy-to-understand explanations.	30/6/1991 0:00:00
BU7832	Straight Talk About Computers	business	1389	19,99	5000	10	4095	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.	22/6/1991 0:00:00
MC2222	Silicon Valley Gastronomic Treats	mod_cook	0877	19,99	0	12	2032	Favorite recipes for quick, easy, and elegant meals.	9/6/1991 0:00:00
MC3021	The Gourmet Microwave	mod_cook	0877	2,99	15000	24	22246	Traditional French gourmet recipes adapted for modern microwave cooking.	18/6/1991 0:00:00
MC3026	The Psychology of Computer Cooking	UNDECIDED	0877						6/8/2000 1:33:54
PC1035	But Is It User Friendly?	popular_comp	1389	22,95	7000	16	8780	A survey of software for the naive user, focusing on the 'friendliness' of each.	30/6/1991 0:00:00
PC8888	Secrets of Silicon Valley	popular_comp	1389	20	8000	10	4095	Muckraking reporting on the world's largest computer hardware and software manufacturers.	12/6/1994 0:00:00
PC9999	Net Etiquette	popular_comp	1389					A must-read for computer conferencing.	6/8/2000 1:33:54
	Computer Phobic							A must for the specialist, this	

Listo Intranet local

Figura 21.4. Podemos consultar el contenido de la tabla desde cualquier navegador a través de Internet

El siguiente fragmento de código, por ejemplo, añadiría un nuevo autor a la tabla `authors`.

```
' Abrir conexión con la base de datos
Dim Conexion As SqlConnection = New SqlConnection( _
   "server=INSPIRON;uid=sa;pwd=;database=pubs")

' Llenar un DataAdapter con la tabla
Dim Comando As SqlDataAdapter = New _
   SqlDataAdapter("SELECT * FROM Authors", Conexion)

' Creamos un DataSet local
Dim MiDataSet As DataSet = New DataSet()

' Llenamos con la tabla
Comando.Fill(MiDataSet, "Autores")
' y recuperamos una referencia a ella
Dim MiTabla As DataTable = MiDataSet.Tables("Autores")

' Creamos un nuevo registro
Dim NuevaFila As DataRow = MiTabla.NewRow()
```

```
' Establecemos los valores
NuevaFila("au_id") = "123-45-6789"
NuevaFila("au_lname") = "Charte"
NuevaFila("au_fname") = "Francisco"
NuevaFila("phone") = "999 222 333"
NuevaFila("address") = "Apdo. 54"
NuevaFila("city") = "Jaén"
NuevaFila("zip") = "23080"
NuevaFila("contract") = False

' La añadimos a la tabla
MiTabla.Rows.Add(NuevaFila)
```

En este momento la nueva fila se encuentra en el `DataSet`, pero no en la base de datos de la que se han recuperado previamente las demás filas. Tenemos dos opciones: asignar manualmente a la propiedad `InsertCommand` la sentencia SQL para efectuar la inserción de la nueva fila o, dado que la actualización sólo afecta a una tabla, usar un componente `SqlCommandBuilder` para generar automáticamente los comandos de actualización, inserción y borrado. Utilizando este segundo sistema bastarían las dos sentencias siguientes, detrás del código anterior, para ejecutar la inserción de la fila en el servidor de datos.

```
' Generamos los comandos de actualización
Dim Comandos As SqlCommandBuilder = _
  New SqlCommandBuilder(Comando)

' Actualizamos
Comando.Update(MiDataSet, "Autores")
```

En la figura 21.5 puede ver la tabla tras efectuar la operación. Para obtener este resultado se ha añadido al mismo formulario un botón y modificado el código para enumerar las filas de la tabla de autores en lugar de la de títulos.

Nota

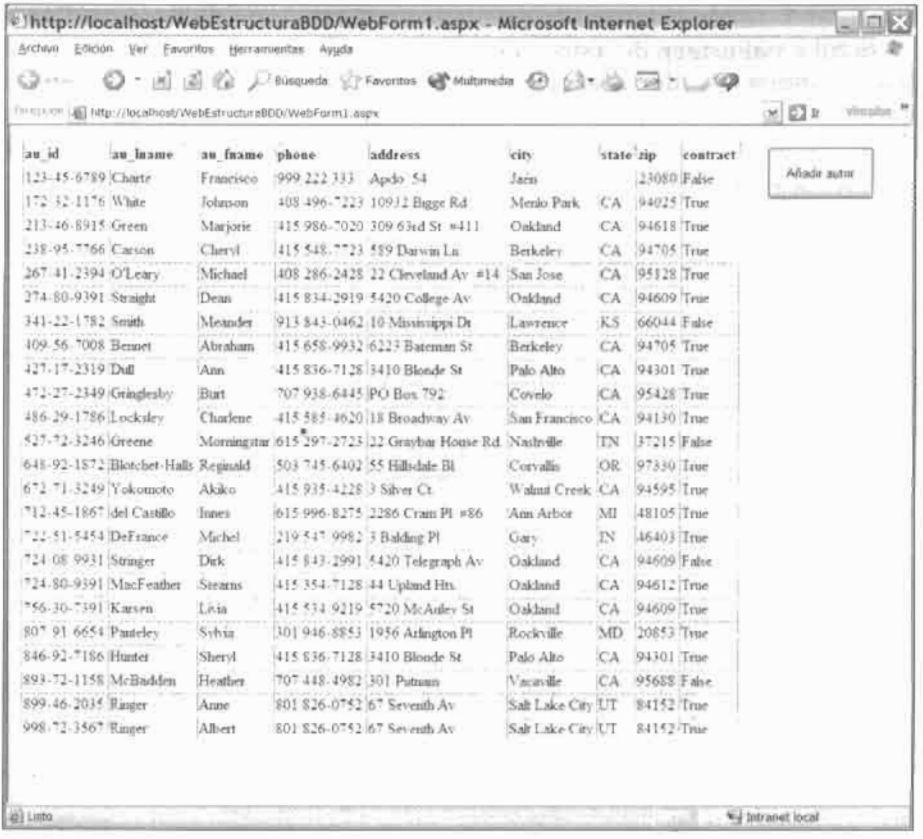
Si pulsa dos veces el botón para añadir verá que obtiene un error, concretamente una violación de la clave primaria, ya que no pueden existir dos autores con el mismo código.

Enlace de datos e interfaz

En los puntos previos hemos estado accediendo a los datos desde el código de nuestros ejemplos, mostrando la información o modificándola. La mayoría de las aplicaciones, sin embargo, no efectúan estas tareas de forma directa, sino ofreciendo a los usuarios una interfaz que les facilite la visualización, modificación, eliminación e inserción de datos.

Obviamente podríamos, siguiendo la técnica utilizada anteriormente en dichos ejemplos, recuperar los datos, asignarlos como contenido de controles,

permitir que el usuario se desplace y modifique y, finalmente, ejecutar manualmente los métodos que procedan. Es mucho más fácil, sin embargo, utilizar lo que se conoce como enlace entre controles de interfaz y datos o *data-binding*.



au_id	au_lname	au_fname	phone	address	city	state	zip	contract
123-45-6789	Charte	Francisco	999 222 333	Apdo 54	Jaén		23080	False
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd	Menlo Park	CA	94025	True
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St #411	Oakland	CA	94618	True
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln	Berkeley	CA	94705	True
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av #14	San Jose	CA	95128	True
274-80-9391	Straight	Dean	415 834-2919	5420 College Av	Oakland	CA	94609	True
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr	Lawrence	KS	66044	False
409-56-7008	Bennet	Abraham	415 658-9932	6223 Bateman St	Berkeley	CA	94705	True
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde St	Palo Alto	CA	94301	True
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	95428	True
486-29-1786	Locksley	Charlene	415 585-4620	18 Broadway Av	San Francisco	CA	94130	True
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar House Rd	Nashville	TN	37215	False
648-92-1872	Blotchet-Halls	Reginald	503 745-6402	55 Hillsdale Bl	Corvallis	OR	97330	True
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct	Walnut Creek	CA	94595	True
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl #86	Ann Arbor	MI	48105	True
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl	Gary	IN	46403	True
724-08-9931	Stringer	Dirk	415 843-2991	5420 Telegraph Av	Oakland	CA	94609	False
724-80-9391	MacFeather	Stearns	415 354-7128	44 Upland Hts	Oakland	CA	94612	True
756-30-7391	Karsen	Livia	415 534-9219	5720 McAuley St	Oakland	CA	94609	True
807-91-6654	Panteley	Sylvia	301 946-8853	1956 Arlington Pl	Rockville	MD	20853	True
846-92-7186	Hunter	Sheryl	415 836-7128	3410 Blonde St	Palo Alto	CA	94301	True
893-72-1158	McBadden	Heather	707 448-4982	301 Putnam	Vacaville	CA	95688	False
899-46-2035	Ringer	Anne	801 826-0752	67 Seventh Av	Salt Lake City	UT	84152	True
998-72-3567	Ringer	Albert	801 826-0752	67 Seventh Av	Salt Lake City	UT	84152	True

Figura 21.5. La tabla de autores tras la operación de inserción

A pesar de que el enlace a datos de muchos controles Web y Windows evita tener que efectuar asignaciones entre las columnas y los controles, no automatizan completamente todo el proceso de edición de datos. Esto es especialmente cierto en el caso de las aplicaciones basadas en formularios Web, en las que será preciso habilitar, por ejemplo, medios para la inserción o para la eliminación de filas.

Enlace en aplicaciones Web

Algunas de las clases de controles de servidor existentes en el ámbito `System.Web.UI.WebControls` y que, por tanto, pueden insertarse en formularios Web, disponen de propiedades mediante las cuales es posible establecer un enlace con un origen de datos. El control `ListBox`, por ejemplo, se puede asociar con una determinada columna de una tabla para mostrar sus valores.

Mediante el control `DataGrid` podemos establecer esa asociación con una tabla completa.

La conexión con la base de datos y obtención de un `DataSet` se efectúa siguiendo exactamente el mismo proceso usado en ejemplos previos. Tras eso basta con asignar a la propiedad `DataSource` de los controles web una referencia al origen de datos que, por regla general, será un objeto `DataSet` o `DataView`.

Puede hacer una simple prueba insertando en un formulario Web un componente `DataGrid`, estableciendo las propiedades que afectan al aspecto de la tabla, como se ha hecho en la figura 21.6, y, finalmente, escribiendo el código siguiente en el método correspondiente al evento `Load`:

```
Dim Conexion As SqlConnection = New SqlConnection( _
   "server=INSPIRON;uid=sa;pwd=;database=pubs")

Dim Comando As SqlDataAdapter = New _
   SqlDataAdapter("SELECT * FROM authors", Conexion)

Dim MiDataSet As DataSet = New DataSet()

Comando.Fill(MiDataSet, "Autores")

DataGrid1.DataSource = MiDataSet.Tables("Autores")
DataGrid1.DataBind()
```

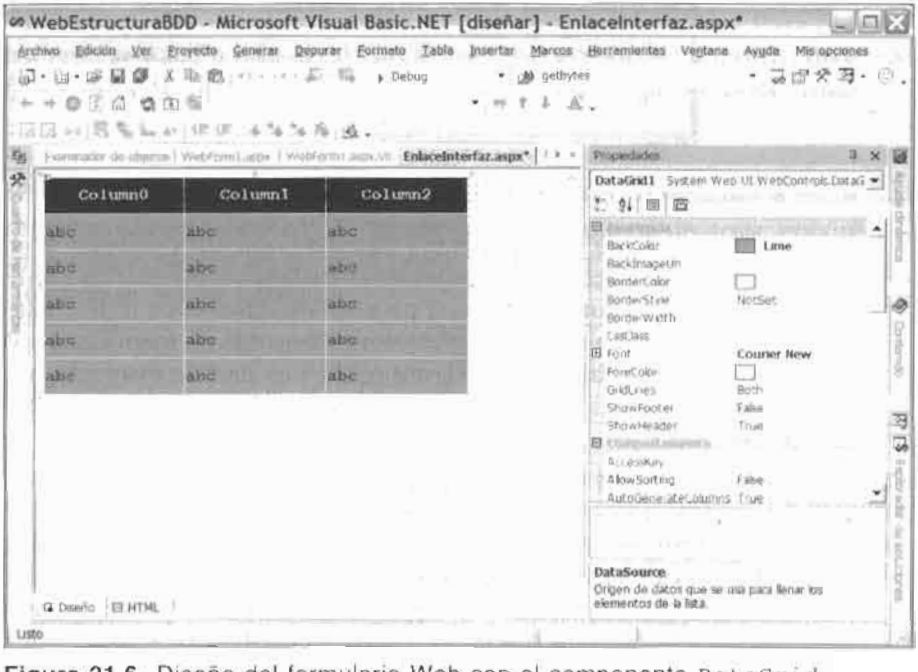
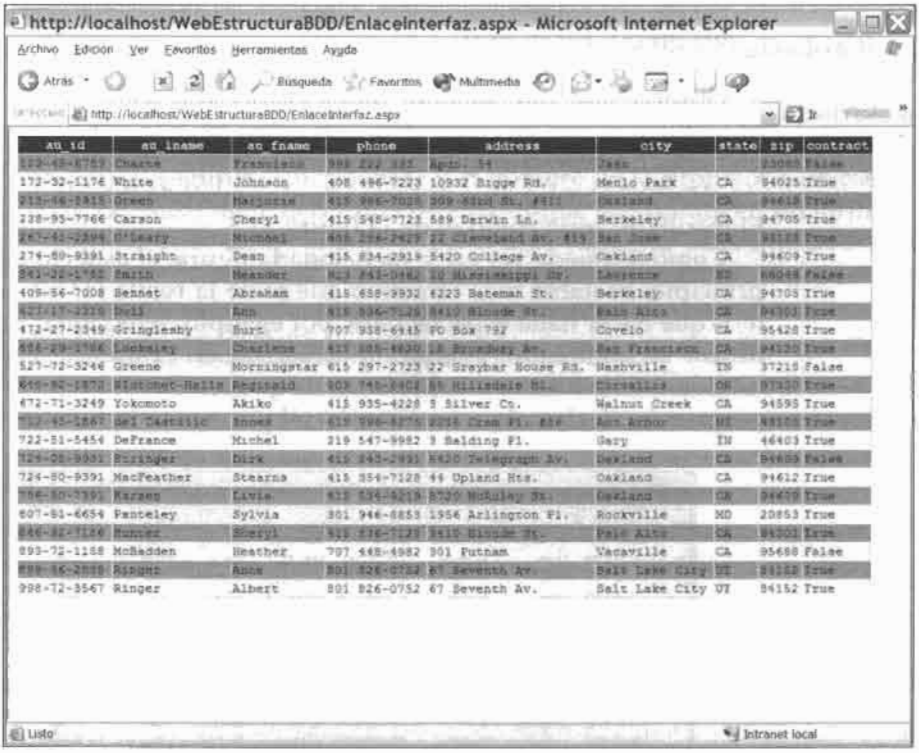


Figura 21.6. Diseño del formulario Web con el componente `DataGrid`

Observe cómo creamos un `DataSet` recuperando la tabla de autores. El último paso es llamar al método `DataBind()` para activar el enlace entre el origen de datos y el control. En la figura 21.7 puede ver el resultado que obtenemos al ejecutar el programa.

Si quisiéramos habilitar la edición de datos en la propia rejilla deberíamos disponer los elementos adecuados, por ejemplo botones, y responder a los eventos generados por la rejilla a medida que se manipula la información.



au_id	au_lname	au_fname	phone	address	city	state	zip	contract
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	94025	True
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	94705	True
274-80-9391	Straight	Dean	415 834-2919	5420 College Av.	Oakland	CA	94609	True
409-56-7008	Bennet	Abraham	415 658-9932	6223 Bateman St.	Berkeley	CA	94705	True
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	46403	True
724-80-9391	MacFeather	Stearns	415 354-7128	44 Upland Hts.	Oakland	CA	94612	True
893-72-1158	McBadden	Heather	707 448-4982	301 Putnam	Vacaville	CA	95688	False
998-72-3567	Ringer	Albert	801 826-0752	67 Seventh Av.	Salt Lake City	UT	84152	True

Figura 21.7. Tabla generada por el componente `DataGrid`

Enlace en aplicaciones Windows

Para construir una aplicación Windows con componentes de interfaz enlazados a datos se utiliza un mecanismo prácticamente idéntico al que acaba de describirse en el punto anterior. Para conseguir una aplicación similar a la página ASP.NET anterior, con una rejilla mostrando el contenido de la tabla de autores, podríamos insertar un componente `DataGrid` en un formulario Windows e introducir el código que puede ver a continuación en el método correspondiente al evento `Load` del formulario.

```
Dim Conexion As SqlConnection = New SqlConnection( _
   "server=INSPIRON;uid=sa;pwd=;database=pubs")
```

```
Dim Comando As SqlDataAdapter = New _
  SqlDataAdapter("SELECT * FROM authors", Conexion)

Dim MiDataSet As DataSet = New DataSet()

Comando.Fill(MiDataSet, "Autores")

DataGrid1.DataSource = MiDataSet
DataGrid1.DataMember = "Autores"
```

Observe que, nuevamente, todo el proceso de conexión y selección de datos es idéntico al que ya conocemos. Lo único adicional es la asignación a las propiedades `DataSource` y `DataMember`, la primera de ellas tomando una referencia al `DataSet` y la segunda tomando el nombre de la tabla que quiere mostrarse. El resultado sería como el de la figura 21.8.

Como se puede ver en la mencionada figura, el aspecto de esta rejilla es, en principio, bastante más atractivo que el de la rejilla HTML. Además, no tenemos que hacer nada especial para, por ejemplo, seleccionar una columna y modificar su contenido, aunque la actualización real en la base de datos no se efectúa automáticamente. Tendríamos que añadir al menos un botón que se encargase de efectuar la actualización tal y como se indicó en un punto previo.

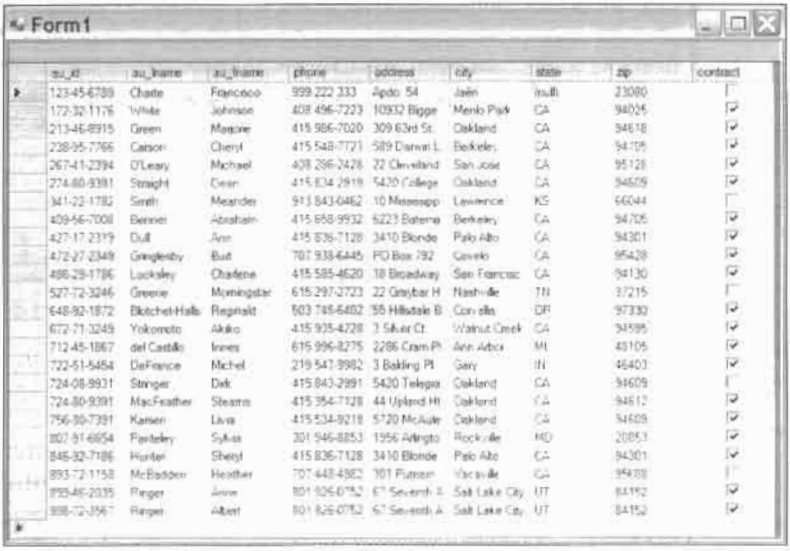


Figura 21.8. El formulario Windows mostrando la tabla de autores

La *colección* `DataBindings`

Además de las propiedades `DataSource` y `DataMember`, de la que disponen algunos componentes que están diseñados específicamente para usarlos

con bases de datos, la mayoría de los controles .NET disponen de una propiedad, llamada `DataBindings`, que facilita el enlace de una cierta propiedad del control con otra propiedad de otro componente. Es un mecanismo muy flexible que puede utilizarse no sólo para conectar una caja de texto con una columna de una tabla, que puede ser lo más usual, sino para enlazar cualquier propiedad de un control con otra propiedad de otro control.

`DataBindings` es una colección de objetos `DataBinding` y, como tal, puede contener uno o más objetos de esa clase. Esto significa que un mismo componente puede contar con varios enlaces de datos.

Como toda colección, `DataBindings` dispone de un método, llamado `Add()`, que facilita la adición de objetos.

Al crear un objeto `DataBinding` deberemos facilitar tres parámetros: la propiedad a enlazar en el primer componente, al que va a añadirse el enlace, una referencia al segundo componente y el nombre de la propiedad de ese segundo componente. Puede hacer una prueba simple insertando en un formulario dos controles `TextBox` y añadiendo el código siguiente al evento `Load` del formulario:

```
Dim Conexion As SqlConnection = New SqlConnection( _
   "server=INSPIRON;uid=sa;pwd=;database=pubs")

Dim Comando As SqlDataAdapter = New _
  SqlDataAdapter("SELECT * FROM authors", Conexion)

Dim MiDataSet As DataSet = New DataSet()

Comando.Fill(MiDataSet, "Autores")

TextBox1.DataBindings.Add("Text", MiDataSet, "Autores.au_fname")
TextBox2.DataBindings.Add("Text", MiDataSet, "Autores.au_lname")
```

Las sentencias que hay al final son las que enlazan las dos cajas de texto con dos de las columnas de la tabla. Al ejecutar el programa verá cómo aparece en esas cajas de texto el nombre y apellido de la primera fila recuperada.

Asistentes y diseñadores de Visual Basic .NET

Algunos de los beneficios de ADO.NET no los podremos apreciar si nos limitamos a crear los `DataSet` y adaptadores desde código, sin usar los asistentes y diseñadores de Visual Basic .NET. Utilizando esta herramienta, cada vez que se seleccione en el diseñador de datos un origen, como puede ser la tabla de autores, se generarán automáticamente todas las clases necesarias para que podamos hacer referencia directa a tablas y columnas como si fuesen objetos.

En ejemplos previos hemos utilizado la sintaxis `Tables("Autores")` para referirnos a una determinada tabla, o `Columns("title")` para poder acceder a una cierta columna. Usando las clases generadas por Visual Basic .NET esas mismas referencias quedarian como `Autores` y `title`, sin necesidad de anteponer el nombre de la colección, los corchetes o las comillas.

Otra ventaja de utilizar estos objetos, aparte de la notación más clara, es que la tecnología *Intellisense* de Visual Basic .NET podrá ser mucho más útil, por ejemplo mostrando todas las columnas al referirse a una tabla o todas las propiedades de una determinada columna.

Creación automática de objetos

Encontraremos en Visual Basic .NET diferentes métodos para conseguir la creación automática de objetos como los usados en ejemplos previos para la conexión, recuperación y acceso a los datos. Uno de los más directos y fáciles consiste en usar el **Examinador de servidores**.

Usando la rama **Conexiones de datos** de esa ventana (véase figura 21.9) es posible acceder a cualquier conexión que haya definida. Use el menú emergente de dicha rama, concretamente la opción **Agregar conexión**, para definir nuevas conexiones. Como se aprecia en la figura 21.9, podemos acceder a las tablas, vistas, procedimientos almacenados y diagramas que hay en la base de datos.

Figura 21.9. Exploramos la conexión con un servidor

Tomando un elemento, como puede ser una tabla o un conjunto de columnas, y arrastrándolo sobre el diseñador, por ejemplo en un formulario Windows, verá cómo se añaden dos elementos no visuales: un `SqlConnection` y un

`SqlDataAdapter`. Nos ahorramos, por tanto, el código de conexión y recuperación de un conjunto de datos que usamos en todos los ejemplos precedentes.

Seleccionando cualquiera de los elementos recién añadidos, podremos acceder a sus propiedades en la ventana Propiedades. De esta manera podríamos, por ejemplo, alterar la conexión a la base de datos o modificar la sentencia de selección.

Una alternativa, si deseamos configurar los componentes de conexión y recuperación de datos de forma manual, consiste en tomarlos de la página Datos del Cuadro de herramientas, mostrada en la figura 21.10, e insertarlos en el diseñador, procediendo como con cualquier otro componente.

Figura 21.10. Componentes relacionados con el acceso a datos

Sirviéndonos de las opciones del menú emergente del adaptador de datos, en este caso concreto el componente `sqlDataAdapter1` que aparece en la parte inferior del diseñador, podemos abrir una ventana para configurarlo, acceder a una vista previa de los datos o generar el correspondiente conjunto de datos o `DataSet`. Gracias a la vista previa podemos conocer anticipadamente la información sobre la que trabajaremos. En la ventana, similar a la de la figura 21.11, aparece en la parte superior una lista de los adaptadores disponibles. Seleccionando uno de ellos y pulsando el botón **Llenar conjunto de datos** veremos las filas y columnas en la parte inferior.

Usando la opción Generar conjunto de datos del citado menú abriremos un asistente, compuesto sólo de una simple ventana, que nos permitirá generar el componente `DataSet` necesario para trabajar con los datos. La disponibilidad de este elemento durante la fase de diseño aporta bastantes ventajas. Para empezar, la propiedad `DataBindings` de cualquier componente despliega, en la ventana Propiedades, una lista de los elementos de base de datos existentes. De esta forma, podemos establecer el enlace adecuado sin necesidad de escribir código alguno, tan sólo eligiendo un elemento de una lista tal y como se muestra en la figura 21.12. Lo mismo es cierto para las propiedades `DataSource` y

`DataMember` de otros componentes, como `DataGrid`, habilitando incluso la configuración de las columnas en fase de diseño.

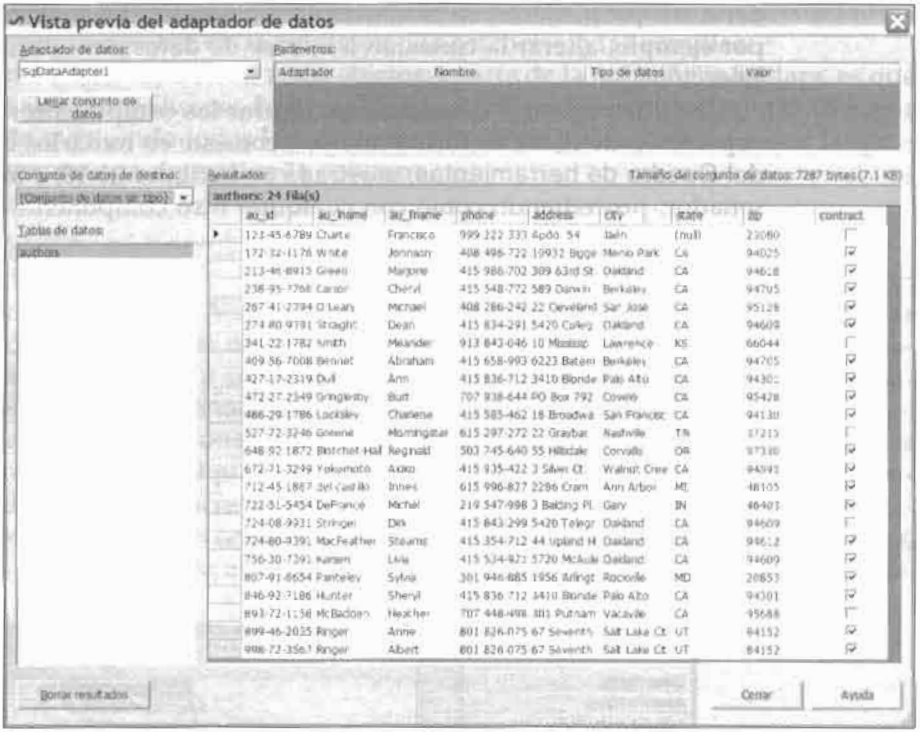


Figura 21.11. Vista preliminar de la información de un adaptador

Figura 21.12. Enlace de controles con datos en la fase de diseño

Otra ventaja considerable es la posibilidad de acceder a los datos, desde el código del programa, sin necesidad de usar propiedades como `Tables` o `Rows`, ya que el asistente que crea el `DataSet` realmente define un derivado en el que existen propiedades que representan a las tablas y columnas con tipos específicos, facilitando la edición y asegurando la ausencia de errores de tipo. En la figura 21.13, por ejemplo, puede ver que el `DataSet` generado por el asistente cuenta con una propiedad, llamada `authors`, que representa a la tabla de autores. Este objeto, a su vez, dispone de una serie de propiedades que representan a las columnas de la mencionada tabla.

Figura 21.13. El conjunto de datos generado dispone de propiedades específicas para acceder a las tablas y columnas

Como puede ver, gracias a los elementos existentes en Visual Basic .NET podemos incluso crear los mismos programas anteriores, de visualización de los datos de una tabla, sin necesidad prácticamente de escribir código.

El asistente para formularios de datos

Para crear un formulario de acceso a datos, ya sea basado en un `DataGrid` o en conjuntos independientes para la edición, podemos usar el asistente específico que existe con este fin. Él se ocupará de añadir un formulario al proyecto e incluir en él todos los elementos necesarios, sin necesidad de configurarlos manualmente. Los pasos a dar son los descritos a continuación.

Añadimos un nuevo elemento al proyecto actual, seleccionando de la lista Plantillas (véase figura 21.14) el elemento Asistente para formularios de datos.

En la parte inferior podemos modificar el nombre establecido por defecto para este nuevo formulario. Aparecerá la primera página del asistente, tan sólo informativa, en la cual pulsaremos el botón **Siguiente**.

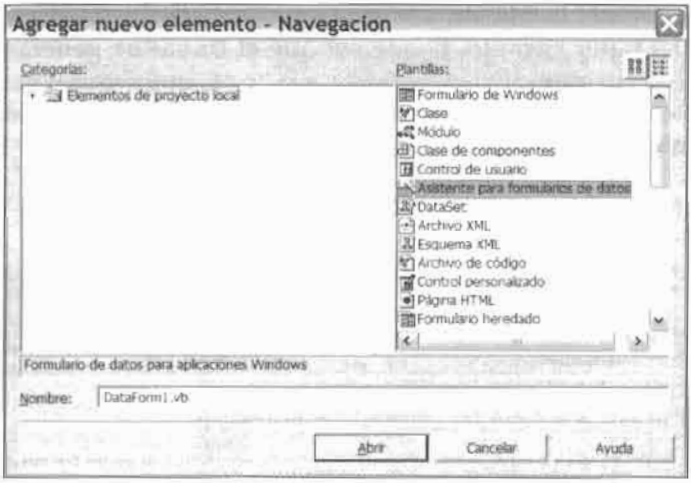


Figura 21.14. Iniciamos el asistente para crear nuestro formulario de datos

El asistente precisa un conjunto de datos sobre el que trabajar, conjunto de datos que podría existir ya, si hubiésemos insertado o creado previamente un `DataSet`, o bien puede ser creado en ese momento. Es la opción por la que se ha optado en la figura 21.15, introduciendo el nombre que va a dársele al nuevo conjunto de datos.

Figura 21.15. Creamos un nuevo conjunto de datos

La siguiente página del asistente, que puede ver en la figura 21.16, sólo aparece en caso de que hayamos elegido la creación de un nuevo `DataSet`. Como puede verse, debemos elegir los elementos que formarán parte de ese conjunto

de datos, seleccionando tablas o vistas y añadiéndolas a la lista de la derecha. De manera análoga, en la página siguiente elegiríamos las columnas que formarían parte del conjunto de datos.

Figura 21.16. Seleccionamos los elementos que formarán parte del conjunto de datos

Finalmente, como último paso del asistente, indicaremos si deseamos mostrar todos los datos en una cuadrícula o rejilla, utilizando un `DataGrid`, o bien usar controles independientes para editar cada columna de una fila cada vez. Si elegimos esta opción, que es lo que ha hecho en la figura 21.17, serán accesibles las opciones Agregar, Eliminar, Cancelar y Controles de exploración que aparecen debajo. Con ellas podremos configurar el formulario según nos interese, añadiendo elementos para la edición y la navegación por los datos.

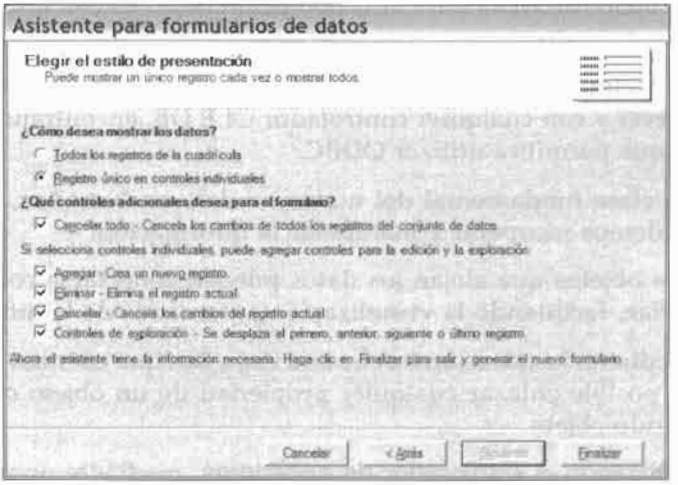


Figura 21.17. Configuramos la apariencia y funciones del formulario de datos

Asumiendo que ha seguido los pasos descritos y seleccionado la misma tabla, columnas y opciones, al finalizar tendrá en su proyecto un formulario similar al de la figura 21.18, mediante el cual podremos efectuar cualquier operación de edición sobre la tabla.

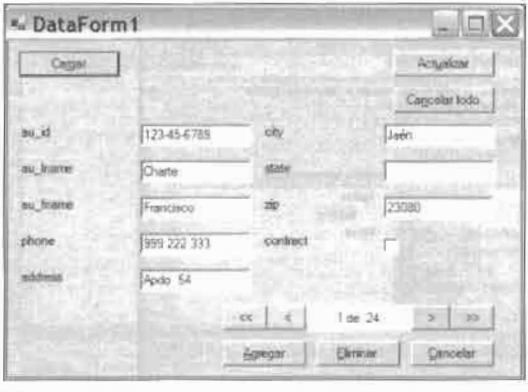


Figura 21.18. Formulario generado por el asistente

No hemos escrito una sola línea de código pero, si examina el módulo correspondiente al formulario de datos, el asistente ha generado bastantes procedimientos. Su estudio puede aclararle muchos detalles sobre objetos y métodos de trabajo con datos.

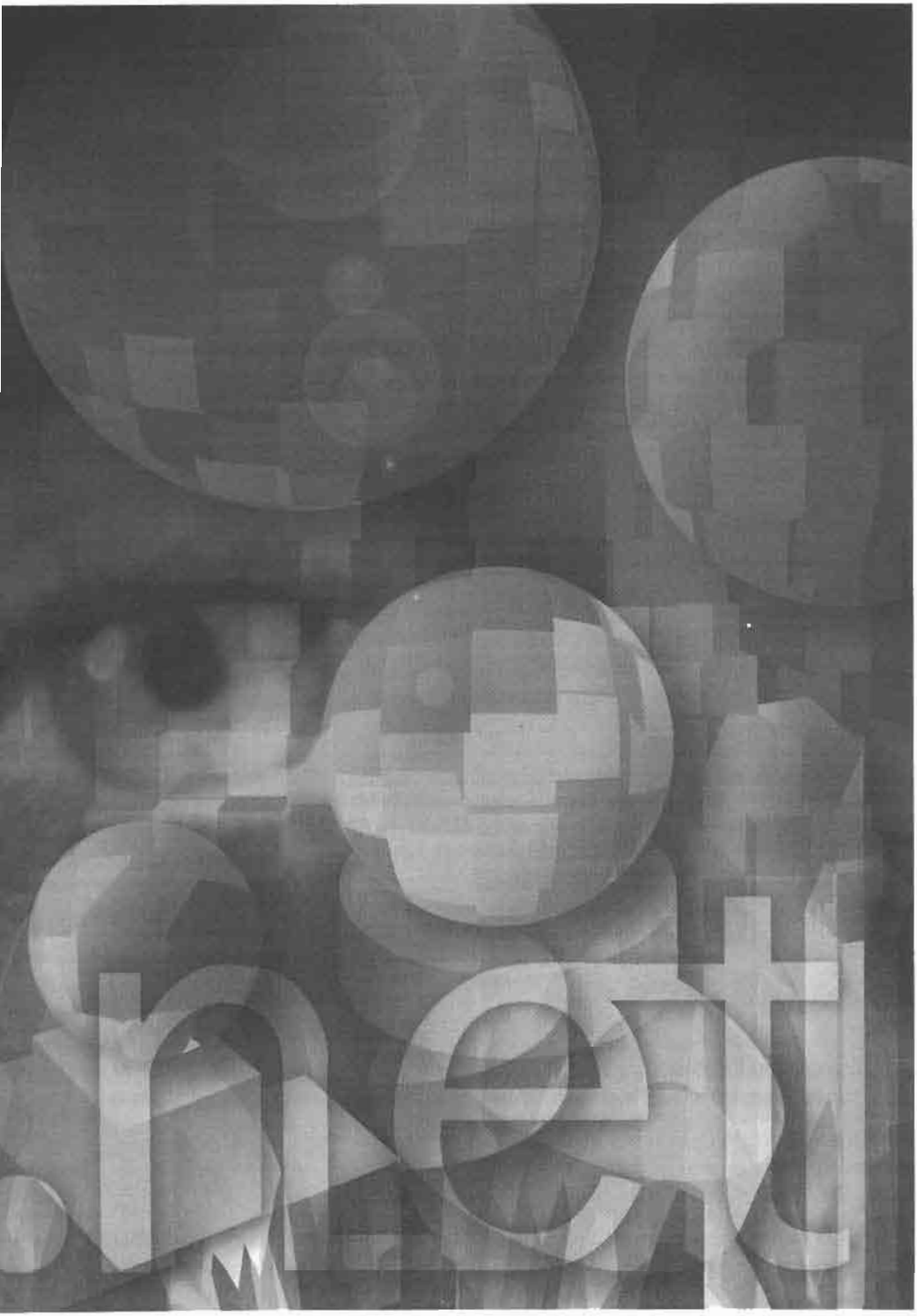
Puntos clave

- Para acceder a bases de datos desde Visual Basic .NET utilizaremos los servicios de ADO.NET, una evolución de ADO con un modelo de objetos simplificado.
- Actualmente existen adaptadores para conectar directamente con SQL Server y con cualquier controlador OLE DB, encontrándose en desarrollo el que permitirá utilizar ODBC.
- La clase fundamental del nuevo modelo es `DataSet`, mediante la cual podemos recuperar y manipular la información.
- Los objetos que alojan los datos pueden conectarse con controles de interfaz, facilitando la visualización automática de la información.
- Mediante la colección `DataBindings` con que cuentan algunos controles es posible enlazar cualquier propiedad de un objeto con otra de un segundo objeto.
- Utilizando el Explorador de servidores, mediante operaciones de arrastrar y soltar, no es necesario definir mediante código las conexiones a bases de datos, comandos y conjuntos de datos.

Resumen

Este capítulo nos ha servido para familiarizarnos con el nuevo mecanismo de acceso a datos que acompañará a la plataforma .NET: ADO.NET. Se trata de una evolución de ADO que incorpora notables mejoras, al tiempo que simplifica el trabajo con datos desde cualquiera de los lenguajes existentes para la plataforma.

Existen muchas posibilidades adicionales que ni siquiera se han mencionado en este capítulo, como por ejemplo la capacidad para trabajar con XML o el uso de transacciones, la creación de modelos de datos mediante las herramientas incorporadas en la edición *Enterprise Architect* de Visual Studio .NET, etc. Lo visto, no obstante, es un buen punto de partida si queremos profundizar en el funcionamiento y uso de ADO.NET y los elementos que Visual Basic .NET incorpora para ello.


.net

22

Diseño de informes con Crystal Reports .NET

En el decimocuarto capítulo se mostró cómo diseñar, prácticamente de forma manual, un documento para imprimirlo o visualizarlo previamente. Esta tarea es factible cuando dicho documento es relativamente simple, pero resulta ardua a medida que se va ganando en complejidad, especialmente si el informe se compone de información obtenida a partir de bases de datos, con las cuales hay que conectar y comunicarse según el proceso descrito en el capítulo previo.

La alternativa, para simplificar ese trabajo, es usar una herramienta específica que facilite el diseño y configuración visual de los informes. Actualmente la única integrada en Visual Basic .NET es Crystal Reports .NET, la cual acompaña a algunas ediciones del producto de Microsoft. Es de suponer que también podrá adquirirse separadamente.

Gracias a Crystal Reports .NET podemos no solo imprimir los informes, sino también publicarlos digitalmente en diversos formatos: HTML, RTF, PDF,

etc., y exponerlos como servicios Web para que cualquier cliente pueda utilizarlos. De esta forma es más fácil la transferencia de los documentos a través de redes empresariales o Internet, en contraposición a las dificultades que plantea el envío de un medio físico como es el papel.

El diseñador de informes

Crystal Reports .NET se integra en el entorno de Visual Studio .NET aportando un asistente y un diseñador visual, así como una colección de objetos que pueden utilizarse para modelar los informes. La información mostrada en éste, los datos propiamente dichos, generalmente se recuperan de una base de datos, aunque en realidad puede actuar como origen cualquier `DataSet` independientemente de dónde provenga su contenido.

Una vez diseñado el informe, podemos imprimirlo directamente, visualizarlo en el interior de un formulario Windows u ofrecerlo en una página web mediante ASP.NET. También existe la posibilidad de ofrecer el informe como si fuese un servicio web. En los puntos siguientes nos vamos a familiarizar con Crystal Reports .NET diseñando un informe simple.

Creación del informe

Asumiendo que tenemos un proyecto al que deseamos añadir un informe, recurriríamos a la ventana **Agregar nuevo elemento** (véase figura 22.1) que hemos utilizado en algún caso previo para añadir formularios y otros elementos. En este caso concreto elegiríamos de la lista de la derecha el elemento **Crystal Report**, introduciendo en la parte inferior el nombre que deseamos dar al archivo donde será almacenado.

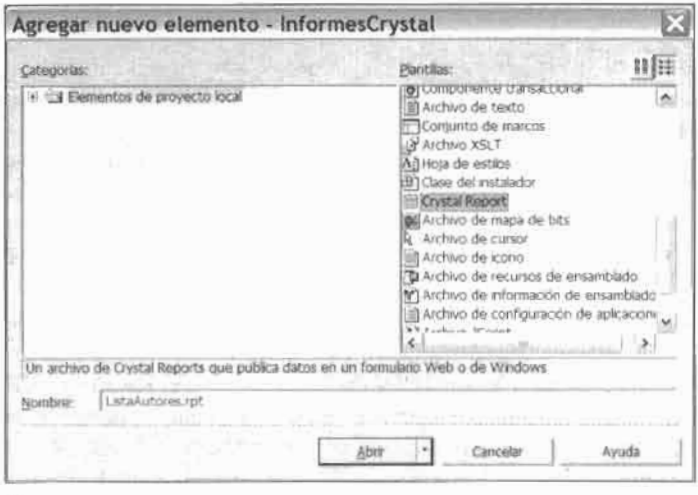


Figura 22.1. Añadimos un informe Crystal Report al proyecto actual

Tenemos tres posibilidades a la hora de crear el informe: utilizar un asistente que nos guíe paso a paso en el proceso, partir de un informe en blanco y añadir a continuación manualmente los elementos que se precisen o bien partir de un informe previo. Asumiendo que es la primera vez que creamos un informe Crystal Reports en Visual Basic .NET, elegiríamos la primera de las opciones disponibles.

Utilizando el asistente, como se ha hecho en la figura 22.2, en la parte inferior podremos elegir entre varios modelos distintos de informes predefinidos. Puede elegir cualquiera de ellos y ver en el dibujo de la derecha cuál sería su apariencia. Después de seleccionar el que más se ajuste a nuestras necesidades, no tenemos mas que pulsar ahora el botón **Aceptar** para poner el asistente en marcha.

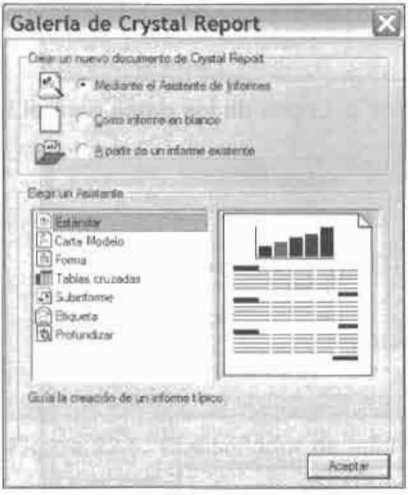


Figura 22.2. Elegimos el tipo de informe a crear

El asistente para la creación de informes dispone, como se aprecia en la figura 22.3, de múltiples páginas, parte de las cuales están dedicadas a la selección del origen de los datos que van a utilizarse, eligiendo los campos o columnas de tablas y estableciendo grupos y totales.

En la mencionada figura, por ejemplo, puede observar cómo se ha conectado con un servidor SQL Server, llamado `INSPIRON`, y se ha tomado de él la tabla `authors` de la base de datos `pubs`.

Algunas de las páginas del asistente serán exclusivas del tipo de informe seleccionado. En la figura 22.4, por ejemplo, puede ver la página para un informe de tipo carta. En dicha página, en la parte superior, existe una lista desplegable para seleccionar la sección del informe que va a diseñar: encabezado de informe, encabezado de página o detalle.

Por cada una de estas secciones tenemos, en la parte inferior, un área para insertar cualquier texto, texto en el que pueden utilizarse elementos de la base de datos tomándolos de la lista que hay encima y arrastrándolos hasta donde se deseen incluir.

Figura 22.3. Seleccionamos el origen de los datos para el informe

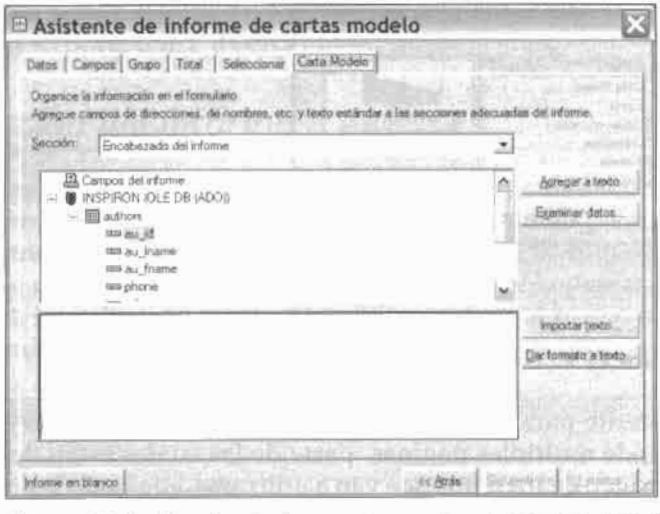


Figura 22.4. Diseño de la secciones de un informe tipo carta

Modificación del diseño

Al cerrar el asistente veremos aparecer el informe en su diseñador, incluyendo en algunas secciones los elementos introducidos por el propio asistente: campos de datos en la sección de detalle, título en el encabezado, etc. Al típico Cuadro de herramientas, en cuyo interior encontramos etiquetas de texto, líneas y recuadros, se añade ahora otra ventana: el Explorador de campos. Puede verlo en la parte inferior izquierda de la imagen que aparece en la figura 22.5.

Gracias a los elementos existentes en el Explorador de campos podrá incluir, en cualquier sección del informe, datos obtenidos del origen previamen-

te seleccionado, por regla general una base de datos, así como fórmulas, el número de página, la fecha de impresión o de confección del informe, etc.

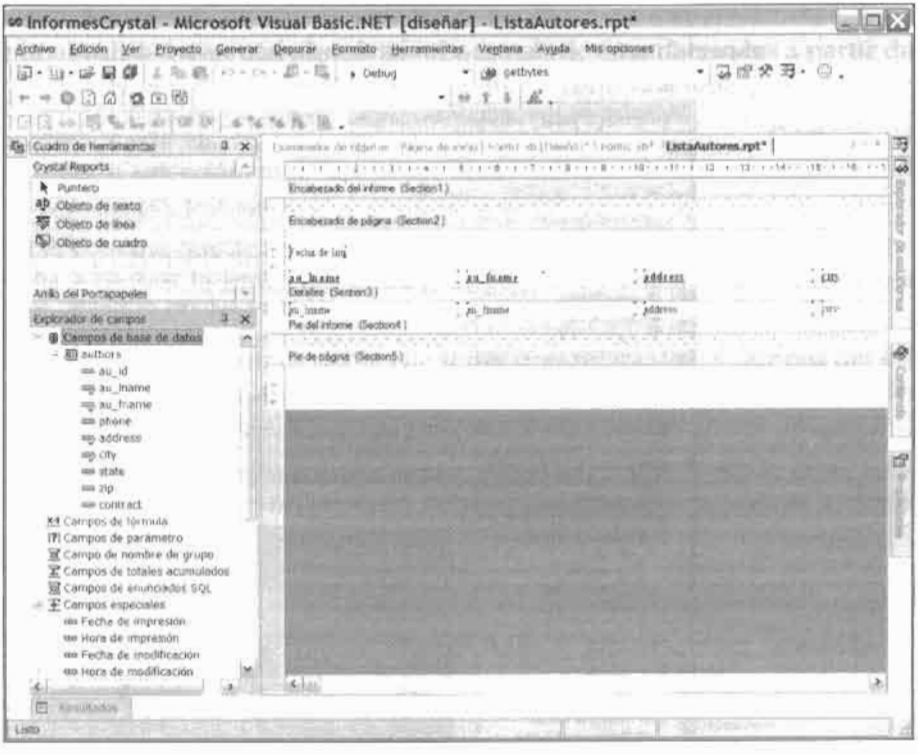


Figura 22.5. El diseñador de informes Crystal Reports

Usando el menú emergente del diseñador, previa colocación del puntero del ratón sobre el lugar en que quiere trabajarse, tendremos acceso a opciones para insertar secciones, subtotales, datos y otros asistentes que simplifican la mayoría de las tareas de diseño. No necesitamos, por tanto, recurrir a ninguna herramienta externa.

En principio la sección de encabezado del informe está inactiva. El sombreado que aparece en su interior indica que no será impresa. Seleccione dicha sección, simplemente pulsando en la zona gris donde pone Encabezado del informe (Section1), y después recurra a la ventana Propiedades para modificar la propiedad `Supress` dándole el valor `False`. A continuación tome del Cuadro de herramientas un Objeto de texto e insértelo en la sección recién activada. Haga doble clic sobre el objeto añadido e introduzca un título, modificando el tipo de letra mediante la habitual propiedad `Font`. Disponiendo el puntero del ratón en el margen inferior de la sección verá que puede modificar su altura.

Lo siguiente será modificar el contenido del encabezado de página, sección que aparece en el diseñador justo debajo de la anterior y que, en el momento de la generación del informe, se muestra una vez por página. Actualmente en esa sección hay un elemento que muestra la fecha de impresión, así como unos

títulos de los datos que van a imprimirse en el cuerpo. Haciendo doble clic sobre esos títulos los modificaremos, puesto que actualmente lo que aparece es el nombre de las columnas en la tabla de autores.

Tras estos cambios el informe tendrá la apariencia de la figura 22.6. Fíjese especialmente en las secciones donde hemos modificado algo.

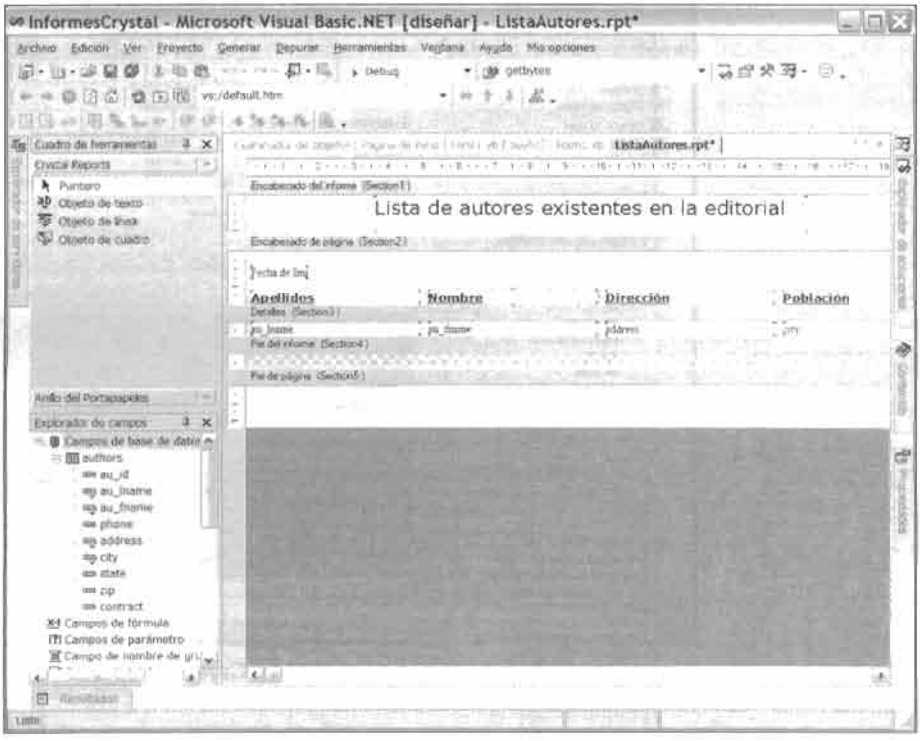


Figura 22.6. El informe tras establecer un título y modificar el encabezado de página

Fórmulas y campos especiales

Además de datos extraídos de una tabla y textos estáticos, que son los componentes básicos que existen actualmente en nuestro informe, también podemos añadir lo que Crystal Reports .NET denomina *campos especiales* y resultados de fórmulas. La lista de campos especiales aparece a la izquierda, en el **Explorador de campos**. De ella podemos tomar la fecha de impresión, el número de páginas que tiene el informe, el nombre de su autor, el número de registro o fila, etc.

Nuestro informe actualmente muestra la fecha de impresión en la parte superior, en el encabezado de página, así como el número de página en el pie de página. Puede alterar tanto la disposición de estos elementos como añadir otros nuevos, por ejemplo mostrando el número total de páginas que van a imprimirse.

La rama Campos de fórmula aparece inicialmente vacía, ya que no existe ninguna fórmula definida en este momento. Suponga que desea añadir al final del informe, en la sección pie del informe que actualmente está desactivada, el número total de autores que hay en la editorial. Con este fin podríamos servirnos de una fórmula muy simple: contar el número de filas obtenidas a partir de la consulta. Los pasos a dar serían los descritos a continuación:

- Pulsamos con el botón secundario del ratón sobre el grupo Campos de fórmula y seleccionamos la opción Nuevo. Introducimos el nombre del nuevo campo, por ejemplo `NumeroAutores`.
- En la ventana que aparece, similar a la de la figura 22.7, seleccionamos de la lista central la función `Count(Campo)`, simplemente haciendo doble clic sobre ella.
- A continuación hacemos doble clic sobre cualquiera de los campos del informe que aparecen en la lista de la izquierda. Cerramos entonces el editor de fórmulas.
- Tomamos el campo `NumeroAutores` y lo arrastramos hasta la sección de pie de informe que, previamente, habremos activado como antes hicimos con el encabezado de informe.

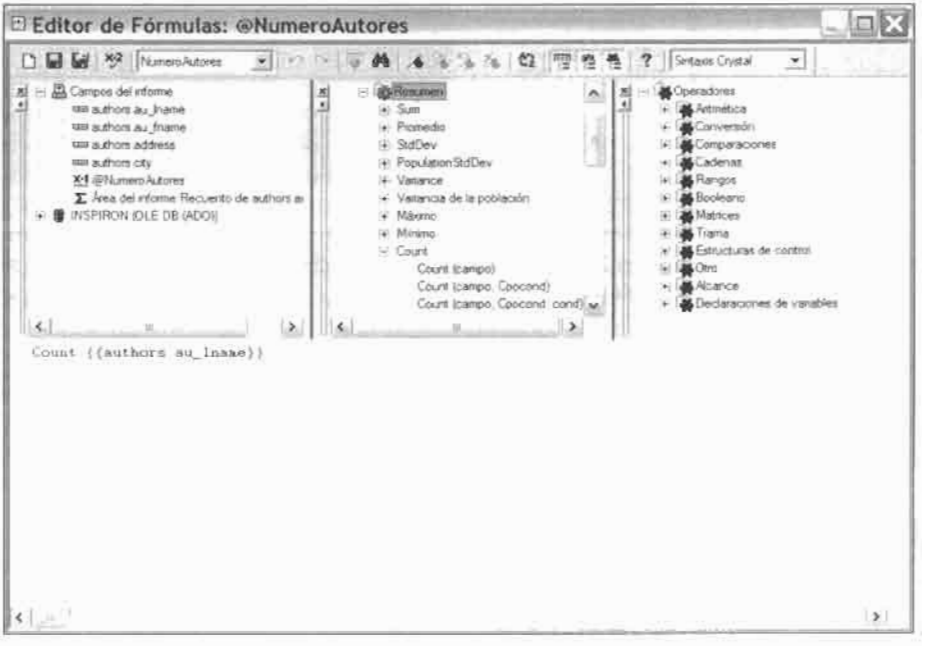


Figura 22.7. El editor de fórmulas de Crystal Reports

Amplíe la sección del pie de informe e incluya un elemento de separación, por ejemplo una línea usando el elemento Objeto de línea que aparece en el Cuadro de herramientas. También puede añadir un texto justo detrás del campo de

fórmula que ha creado indicando dicho número, en este caso autores. En la figura 22.8 aparece el informe tras los últimos cambios. Observe en el margen inferior izquierdo el campo de fórmula que hemos creado y que está disponible para inserción en cualquier punto.

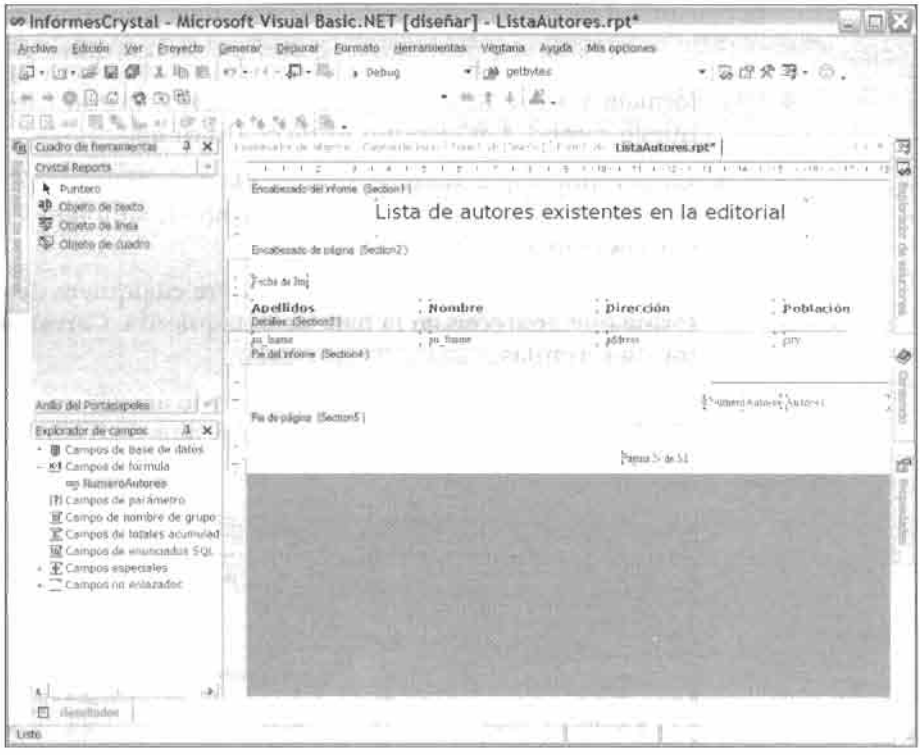


Figura 22.8. El informe tras modificar el pie incluyendo el número de autores

Formato de los elementos

Los elementos que tenemos actualmente en el informe, tanto los dispuestos por el asistente que lo generó como los añadidos posteriormente por nosotros, tienen lo que podríamos denominar un formato por defecto. Ya sabemos que mediante la ventana Propiedades podemos, por ejemplo, alterar el tipo de letra o el color.

Resulta mucho más cómodo editar el formato de cualquier objeto mediante la ventana Editor de formato. El acceso a ésta lo da la opción Formato del menú emergente asociado al objeto en cuestión. Desde esa ventana, según se ve en la figura 22.9, es posible seleccionar el tipo de letra, asociar bordes y sombras, un hipervínculo, rotar el texto, etc. No tenemos más que elegir las opciones que deseemos y, al cerrar la ventana, serán entonces aplicadas al elemento correspondiente.

Figura 22.9. La ventana de edición de formato

Explotación del informe

En los puntos anteriores nos hemos centrado en el diseño del informe, conociendo algunas de las posibilidades que aporta el diseñador de Crystal Reports. Ese diseño, lógicamente, servirá para generar el informe propiamente dicho que es lo que nos interesa. El informe se almacena en un archivo con extensión RPT, archivo desde el cual puede tomarse cuando en una aplicación interese usarlo.

Como se indicaba antes, un informe puede ser impreso directamente, mostrado en un formulario Windows o en un formulario Web. Éstas son algunas de las posibilidades que vamos a ver de inmediato.

Visualización en un formulario Windows

Una vez diseñado el informe, podemos ofrecer una visualización previa en un formulario, vista previa desde la cual el usuario pueda tanto imprimirlo como exportarlo a diversos formatos. Para ello basta con insertar en un formulario un componente `CrystalReportViewer` y enlazarlo con el informe mediante la propiedad `ReportSource`. Los elementos que aparecerán en la ventana de visualización previa pueden controlarse mediante las propiedades `DisplayToolbar`, `DisplayGroupTree`, `ShowExportButton`, `ShowCloseButton`, etc.

Asumiendo que ha iniciado una nueva aplicación Windows, inserte en el formulario un control `CrystalReportViewer` y modifique la propiedad `ReportSource`, localizando el archivo `RPT` en el que habíamos almacenado el informe. El aspecto del control, en la fase de diseño, será el mostrado en la figura 22.10. Hemos dado el valor `False` a las propiedades `DisplayToolBar` y

DisplayGroupTree, de tal forma que en control aparezca tan sólo la vista previa del informe, sin permitir operaciones adicionales.

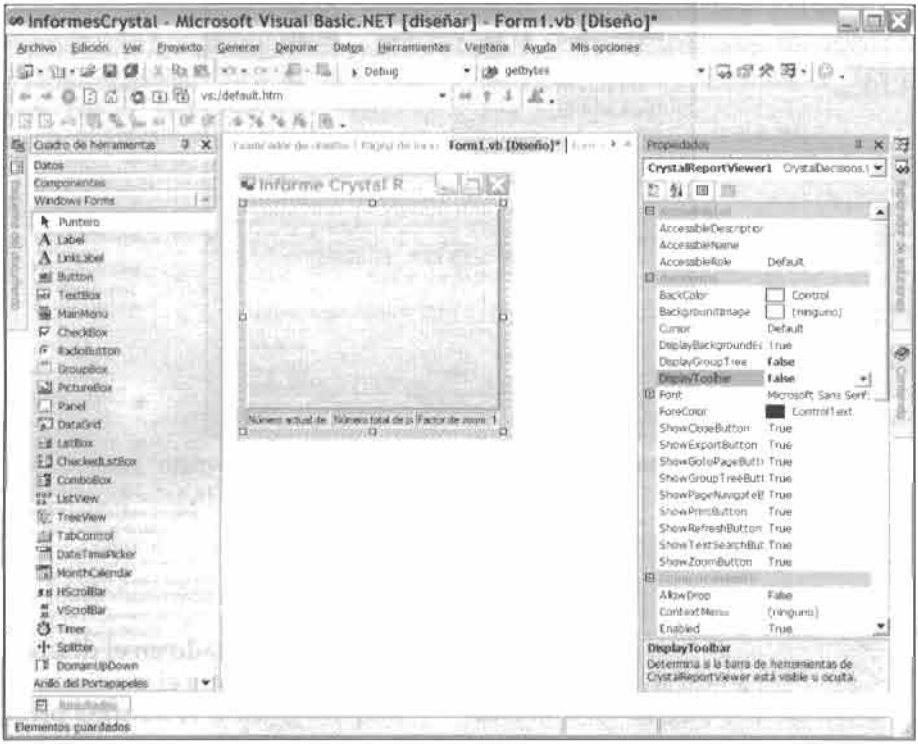


Figura 22.10. El control de vista previa en el interior del formulario

Nota

Si al modificar la propiedad `ReportSource` del control `CrystalReportViewer` ve aparecer una lista tan pequeña que sólo muestra un elemento con el título (Nuevo), pulse la flecha de desplazamiento hacia abajo que aparece a la derecha, en el interior de la misma lista, para poder elegir el elemento Examinar y así recuperar el informe del archivo `RPT`.

Modifique la propiedad `Dock` del `CrystalReportViewer` para que ocupe todo el área disponible en la ventana. No necesita hacer nada más, puede ejecutar el programa para ver el informe resultante que, siguiendo los pasos descritos, debería ser similar al mostrado en la figura 22.11.

Visualización en un formulario Web

Conociendo los pasos que hemos dado para mostrar el informe en un formulario de Windows, no nos costará demasiado hacer lo mismo en un formulario

Web. También contamos en este caso con un control `CrystalReportViewer`, si bien específico para ser incluido en formularios Web. La única diferencia es que en la fase de diseño la propiedad `ReportSource` no aparece en la ventana **Propiedades**, por lo que deberemos recurrir a otro método para acceder al informe. Usaremos el mismo que habíamos diseñado en un punto anterior.

Figura 22.11. Aspecto del informe mostrándose en un formulario Windows

Tras iniciar una nueva aplicación ASP.NET, insertamos en el formulario un componente `CrystalReportViewer` colocándolo aproximadamente en la esquina superior izquierda. Hacemos doble clic sobre el fondo del formulario, para abrir el método asociado al evento `Load`, e introducimos luego el siguiente código:

```
CrystalReportViewer1.ReportSource = _
   "F:\MisInformes\ListaAutores.rpt"
```

Lógicamente, tendrá que modificar el camino para hacer referencia a la carpeta en la que esté almacenado el archivo `ListaAutores.rpt` conteniendo el informe. Puede insertar, si lo desea, otros elementos en el formulario delante o detrás del informe. Sólo con ese componente, no obstante, ya obtendría el resultado que aparece en la figura 22.12. El cliente no sólo ve el informe sino que puede paginarlo, efectuar búsquedas o modificar el factor de magnificación.

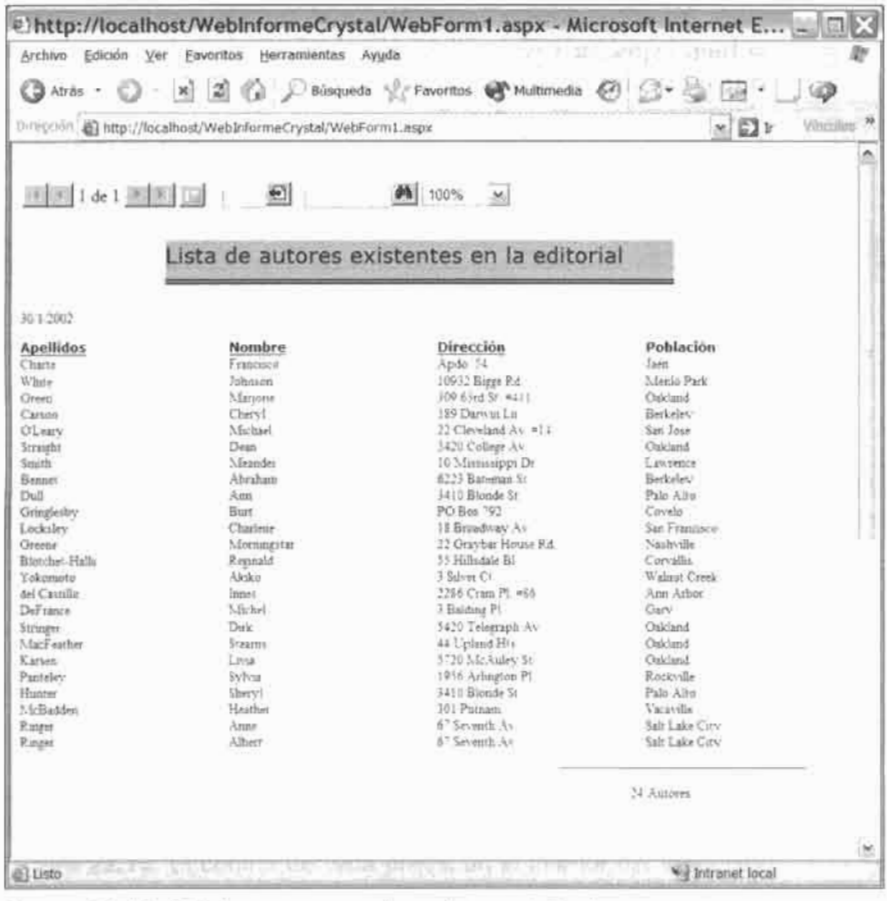


Figura 22.12. El informe mostrado en Internet Explorer

Si desea tener una vista previa del informe durante la fase de diseño, que le permita ajustar algunas propiedades del `CrystalReportViewer`, tendrá que modificar la propiedad `DataBindings`. Ésta da paso a una ventana como la de la figura 22.13, en la que podemos elegir de la izquierda la propiedad que deseamos enlazar, indicando a la derecha la expresión de enlace que, en este caso concreto, será el camino donde se encuentra el informe. Hecho este cambio, ahora podrá ver en el diseñador (véase figura 22.14) cómo quedaría el informe realmente aunque, como se aprecia en la parte central, con unos datos ficticios.

Impresión y exportación del informe

Mostrar el informe visualmente al cliente, ya sea en un formulario Windows o en una página web, puede no ser siempre lo apropiado. En ciertos casos nos interesará simplemente imprimir el informe, o bien guardarlo en un archivo con un cierto formato. No necesitamos usar un control `CrystalReportViewer` para realizar estas operaciones.

Figura 22.13. Modificamos la propiedad `DataBindings` asociando `ReportSource` con el camino donde se encuentra el informe

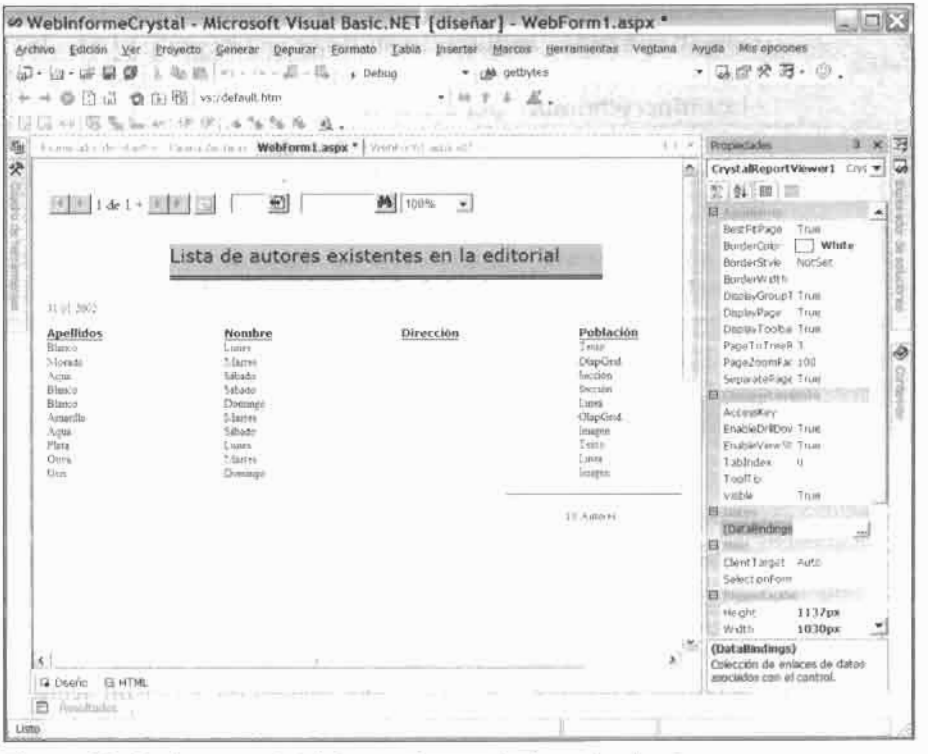


Figura 22.14. Aspecto del informe durante la fase de diseño

Al diseñar un informe estamos creando también una clase. Ésta tendrá por nombre el mismo asignado al informe y, por supuesto, podemos utilizarla para

crear objetos que representan al informe. En la clase encontramos dos métodos especialmente interesantes: `PrintToPrinter()` y `Export()`. Como puede suponer, utilizaremos el primero para enviar el informe a la impresora y el segundo para exportarlo a un archivo.

El método `PrintToPrinter()` precisa cuatro parámetros: un entero indicando el número de copias a imprimir, un `Bolean` indicando si se desean o no las copias en secuencia y dos enteros seleccionando las páginas a imprimir. Si estos dos últimos parámetros son cero se entiende que debe imprimirse el informe completo. Antes de llamar a `PrintToPrinter()` podemos usar la propiedad `PrintOptions` para establecer los parámetros de impresión, por ejemplo seleccionando la orientación del papel, los márgenes, la impresora, etc.

En cuanto al método `Export()`, no es necesario parámetro alguno ya que se asume que, previamente, se habrá utilizado la propiedad `ExportOptions` para configurarlos adecuadamente. Dicha propiedad es un objeto que cuenta con otras propiedades y nos interesarán, principalmente, las que determinan el formato en el que va a exportarse y el destino de la exportación. Mediante `ExportFormatType` elegiremos el formato asignando una de las constantes enumeradas en la tabla 22.1. Las opciones, como puede ver, cubren los casos más habituales.

A continuación asignaremos a la propiedad `ExportDestinationType` el valor `DiskFile`, facilitando en `DestinationOptions` la referencia a un objeto `DiskFileDestinationOptions` que contendrá en su única propiedad el camino y nombre del archivo.

Nota

También existe la posibilidad de exportar un informe enviándolo a una dirección de correo electrónico o una carpeta de Microsoft Exchange.

Tabla 22.1. Formatos de exportación posibles

Constante	Tipo del documento
`Excel`	Hoja de cálculo de Microsoft Excel
`HTML32`	Página HTML ajustada a la especificación 3.2
`HTML40`	Página HTML ajustada a la especificación 4.0
`PortableDocFormat`	PDF
`RichText`	RTF
`WordForWindows`	Documento de Microsoft Word

Suponiendo que deseásemos facilitar una opción para exportar el documento en formato PDF, facilitando así una distribución digital no editable, podríamos disponer un botón en el formulario y asociar el código siguiente al evento de pulsación:

```
Dim MiInforme As New ListaAutores()

Dim Opciones As New DiskFileDestinationOptions()

MiInforme.ExportOptions.ExportFormatType = _
     ExportFormatType.PortableDocFormat

MiInforme.ExportOptions.ExportDestinationType = _
     ExportDestinationType.DiskFile

Opciones.DiskFileName = "C:\MiInforme.pdf"
MiInforme.ExportOptions.DestinationOptions = Opciones

MiInforme.Export()

MessageBox.Show("Informe exportado")
```

En la figura 22.15 se puede ver el mismo informe pero abierto con Acrobat Reader, la utilidad habitual para leer archivos PDF en Windows.

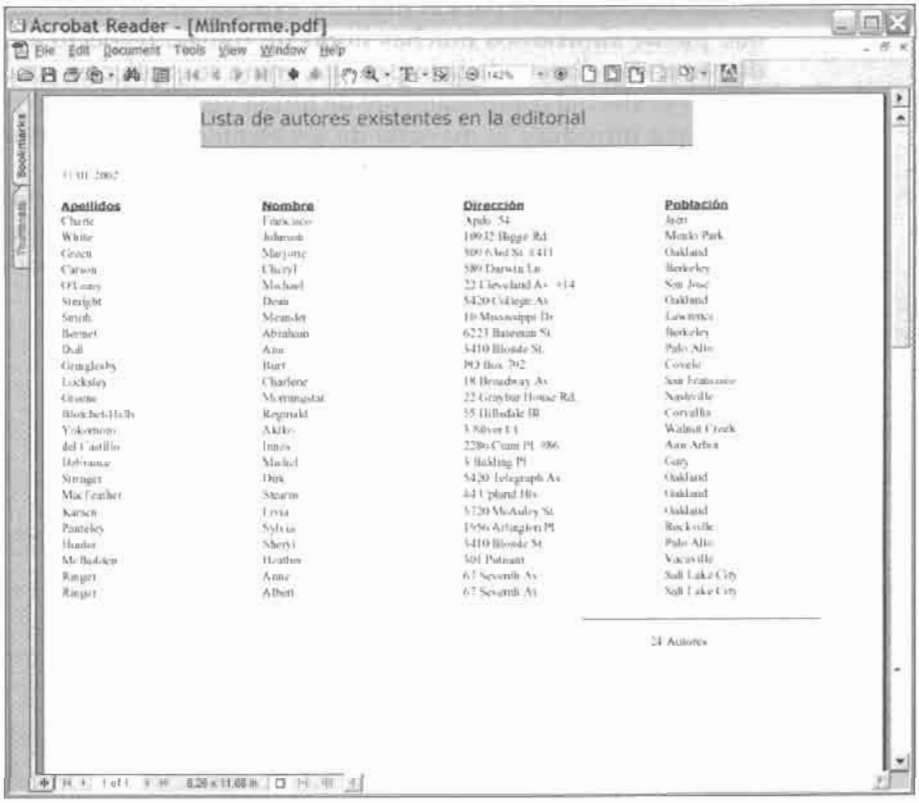


Figura 22.15. El informe en formato PDF

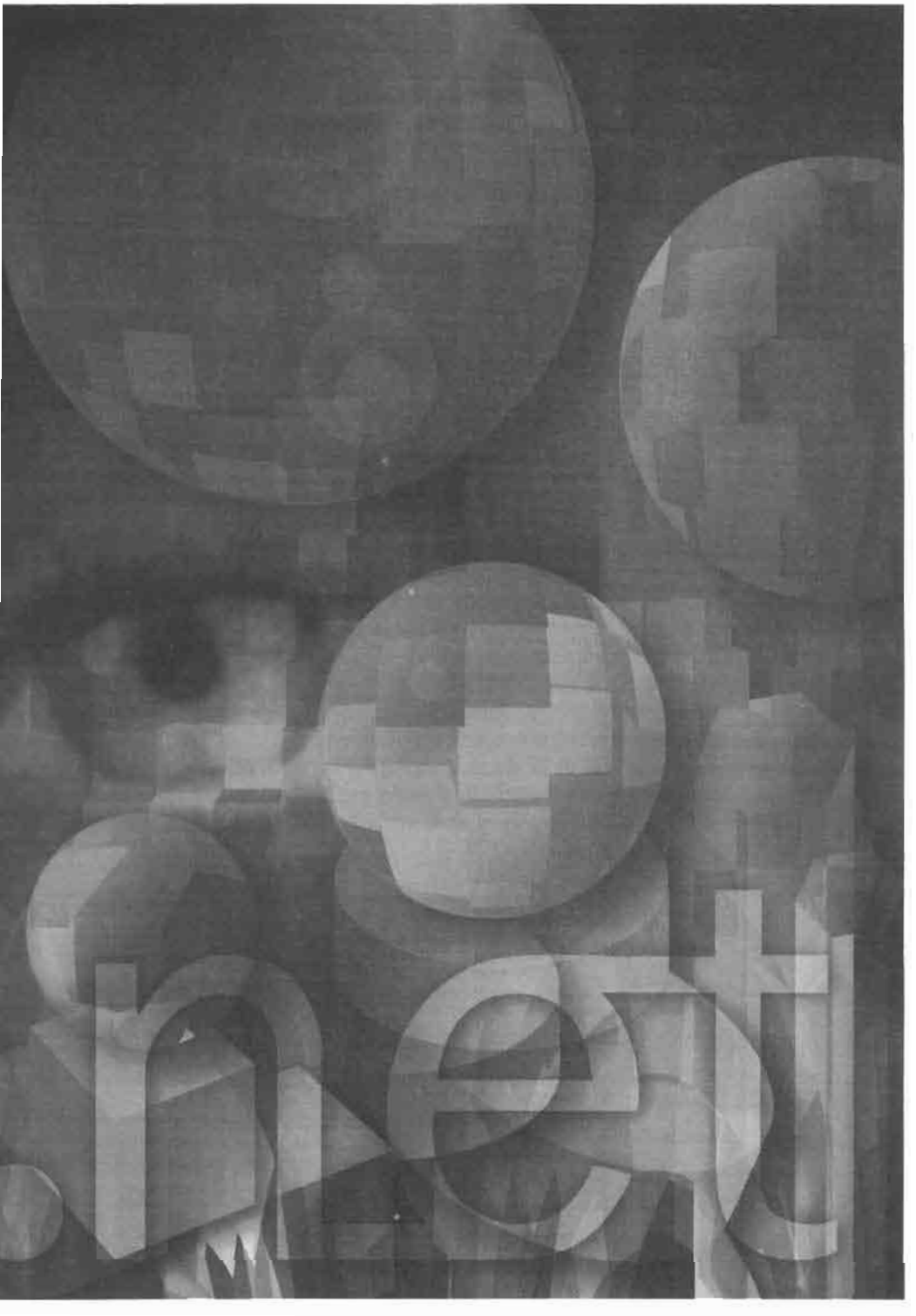
Puntos clave

- Crystal Reports .NET es una herramienta integrada en el entorno de Visual Studio .NET, aportando un diseñador visual para componer los informes y un asistente que los genera a partir de plantillas prediseñadas.
- Usando los objetos de Crystal Reports es posible añadir diversos elementos a un informe, desde texto estático hasta campos especiales y fórmulas.
- Mediante el componente `CrystalReportViewer` es posible facilitar una vista previa en pantalla de un informe, tanto en formularios Windows como en formularios Web.
- El informe genera una clase de componente que podemos usar para crearlo directamente e imprimirlo o exportarlo sin una visualización previa.

Resumen

Como ha quedado patente en este capítulo, Crystal Reports .NET es una herramienta magnífica para el diseño y explotación de informes, una herramienta que puede ahorrarnos muchas horas de trabajo respecto a la publicación mediante el uso directo de la impresora, como se explicó en un capítulo previo. El diseño se efectúa completamente de forma visual, partiendo primero de un asistente que introduce la mayoría de los elementos y, posteriormente, arrastrando nuevos objetos y editando propiedades.

Teniendo un informe diseñado podemos tanto imprimirlo como visualizarlo en un formulario Windows o en una página Web, así como exportarlo a diversos formatos de archivo. Para obtener la misma funcionalidad sin esta herramienta sería necesario invertir muchas horas de trabajo.



23

Recuperación de información de tipos en ejecución

A medida que escribimos código en el editor de Visual Basic .NET nos hemos acostumbrado a contar con listas de miembros de los objetos, al disponer el nombre de un objeto y el punto, o información sobre la lista de parámetros de un método, al abrir unos paréntesis, e, incluso, listas de posibles valores para una variable o propiedad. Todo esto es posible gracias a que en los ensamblados no se almacena tan sólo el código ejecutable, sino que también existe una información adicional sobre tipos que, habitualmente, es conocida como *metadata*.

La meta información, o información tipos, es generada automáticamente para todos los tipos de datos en el momento de la compilación. Parte de esa información es intrínseca al propio tipo de dato: enumeración, estructura, clase, etc., siendo posible añadir otra mediante el uso de atributos. En capítulos previos hemos utilizado atributos para indicar características concretas de un componente o un servicio Web.

Nuestro objetivo, en este capítulo, es aprender a recuperar esa meta información durante la ejecución de un programa. La mayoría de los programas no necesitan dicha información, pero puede ser muy útil en ciertos casos, especialmente cuando una aplicación va a operar sobre elementos cuyas características desconoce de antemano. El diseñador de formularios Windows, por poner un ejemplo que ya conocemos suficientemente, no tiene posibilidad alguna de conocer las propiedades y eventos de todos los componentes que pudieran desarrollarse y, en la práctica, no lo necesita, puesto que puede obtener esa información cuando la necesita. Eso es lo que nos interesa saber a nosotros, cómo obtenerla cuando sea precisa.

Servicios de reflexión

A la hora de obtener información sobre tipos en ejecución nos tendremos que servir de los servicios de reflexión de la plataforma .NET, cuyas clases encontraremos en el ámbito `System.Reflection`. En este ámbito encontramos clases que podríamos denominar de primer nivel, como `Module` o `Assembly`, y otras cuya finalidad es ofrecer información sobre un determinado elemento, como `MemberInfo`, `ParameterInfo` o `EventInfo`. Otra clase fundamental en este contexto, aunque se encuentra en el ámbito `System` al ser básica para la plataforma, es `Type`.

Ya sabemos que todos los objetos con los que tratamos en Visual Basic .NET están, directa o indirectamente, derivados de `System.Object`, incluso los tipos de datos que almacenan un valor y no una referencia. La clase `Object` dispone, entre otros, de un método llamado `GetType()` que devuelve un objeto `Type` con información sobre el objeto en cuestión. Así, es fácil recuperar información a partir de una variable o una propiedad, por ejemplo.

Vamos a comenzar, no obstante, en un nivel bastante más alto que el de las variables o propiedades que pueda contener un objeto, concretamente al nivel de los ensamblados y los módulos que los componen. Después iremos descendiendo rama a rama hasta llegar al detalle máximo como puede ser la lista de parámetros de un método.

Aunque en este capítulo vamos a ocuparnos principalmente de los servicios que nos permiten recuperar información sobre los tipos, usando ciertos servicios del ámbito `System.Reflection` podemos también crear esa información. Son los servicios utilizados por los compiladores y otras herramientas.

Ensamblados y módulos

Cada vez que compilamos un proyecto en Visual Basic .NET estamos generando un ensamblado, generalmente compuesto de un solo módulo que puede ser un ejecutable o una biblioteca. Las propiedades de ese ensamblado, y las acciones que pueden efectuarse sobre él, están representadas por un objeto de

la clase `Assembly`, alojada en el ámbito `System.Reflection`. Normalmente obtendremos un objeto de esa clase mediante dos mecanismos distintos: el método compartido `GetExecutingAssembly()`, que nos entrega el objeto que representa al ensamblado en el que se encuentra el código que está ejecutándose, o bien los métodos `Load()` y `LoadFrom()`, también compartidos, que nos permiten abrir cualquier ensamblado obteniendo un objeto `Assembly` con su información.

Nota

La palabra `Assembly` es una palabra clave en Visual Basic .NET y, por tanto, entra en conflicto con la clase `Assembly`. Para evitar problemas deberá hacer referencia al tipo incluyéndolo entre corchetes: `[Assembly]`.

Disponiendo de un objeto `Assembly`, recuperado con cualquiera de los métodos anteriores, podemos obtener información diversa acerca de él leyendo ciertas propiedades. También podemos enumerar todos los módulos que lo componen, como veremos de inmediato.

Información sobre el ensamblado

Un ensamblado tiene un nombre, una localización física, un punto de entrada, etc. Todos éstos son datos que podemos recuperar sirviéndonos de las propiedades de la clase `Assembly`. Para verlo en la práctica nos serviremos de una sencilla aplicación de consola en la que incluiremos el código siguiente:

```
Imports System.Reflection

Module Module1
  Sub Main()

    Dim EsteEnsamblado As [Assembly] = _
        [Assembly].GetExecutingAssembly


    Console.WriteLine("Nombre del ensamblado: {0}", _
        EsteEnsamblado.FullName)
    Console.WriteLine()
    Console.WriteLine("Localizacion fisica: {0}", _
        EsteEnsamblado.Location)
    Console.WriteLine()
    Console.WriteLine("CodeBase: {0}", EsteEnsamblado.CodeBase)
    Console.WriteLine()
    Console.WriteLine("Está en la cache de ensamblados: {0}", _
        EsteEnsamblado.GlobalAssemblyCache)
    Console.WriteLine()
    Console.WriteLine("EntryPoint: {0}", EsteEnsamblado.EntryPoint)
```

Tras obtener una referencia al objeto `Assembly` que representa al ensamblado actual, el que contiene el código que está ejecutándose, usamos las propiedades `FullName`, `Location`, `CodeBase`, `GlobalAssemblyCache` y `EntryPoint` para recuperar el nombre del ensamblado, su localización física, saber si está o no en la caché global de ensamblados del sistema y conocer el punto de entrada que se ejecuta al iniciar el ensamblado. El resultado obtenido, al ejecutar este programa, será similar al de la figura 23.1. En ella puede ver que el nombre completo incluye la versión del ensamblado, así como que éste no se encuentra en la GAC (*Global Assembly Cache*) y que el punto de entrada es el método `Main()`.

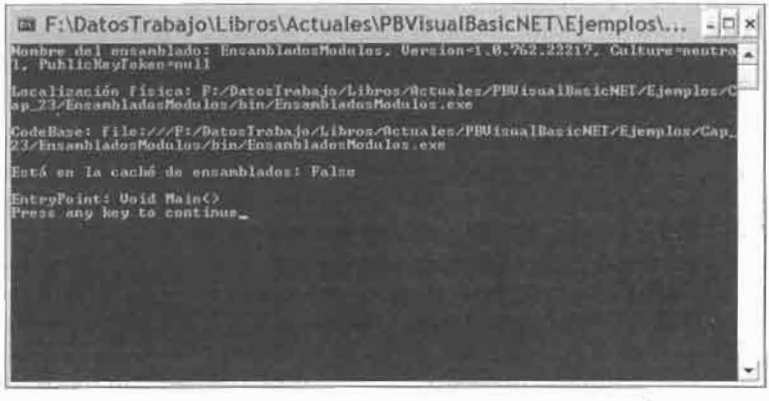


Figura 23.1. Información sobre el ensamblado que contiene el propio ejemplo

Si en lugar de sobre el ensamblado que contiene el ejemplo desea información sobre otro cualquiera, no tiene más que añadir la línea siguiente antes de comenzar la recuperación de datos:

```
EsteEnsamblado = [Assembly].Load("System.Windows.Forms")
```

En este caso se recupera el ensamblado `System.Windows.Forms`, pero de manera análoga podría acceder a cualquier otro ya sea de la plataforma .NET o de terceros. El resultado, como se aprecia en la figura 23.2, es bien distinto. Este ensamblado sí se encuentra alojado en la GAC y no tiene un punto de entrada ya que es una biblioteca, no ejecutable directamente.

Módulos que componen el ensamblado

La clase `Assembly` dispone de varios métodos que nos permiten obtener un cierto módulo de los que componen el ensamblado o bien una lista de todos ellos o todos los que en dicho momento están alojados en memoria. Cada módulo es representado mediante un objeto de la clase `Module` que, al igual que `Assembly`, *es una palabra reservada ya que se usa para la definición de módulos*, de ahí que deba utilizarse también la notación especial con los corchetes.

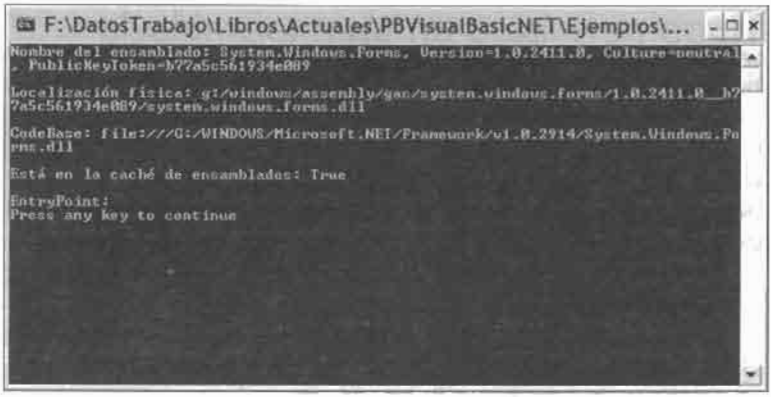


Figura 23.2. Información sobre el ensamblado `System.Windows.Forms`

En la clase módulo tan sólo encontramos cuatro propiedades con información sobre el módulo. Con ellas podemos conocer el nombre simple o cualificado, así como obtener el `Assembly` al que pertenece el módulo. No necesitamos más para, simplemente, enumerar los módulos que tiene un ensamblado conociendo su nombre. Podemos añadir las tres sentencias siguientes al final del código del ejemplo anterior para conseguir un resultado como el de la figura 23.3. En este caso el ensamblado está compuesto de un solo módulo.

```
Dim UnModulo As [Module]
For Each UnModulo In EsteEnsamblado.GetModules()
  Console.WriteLine(UnModulo.Name)
Next
```

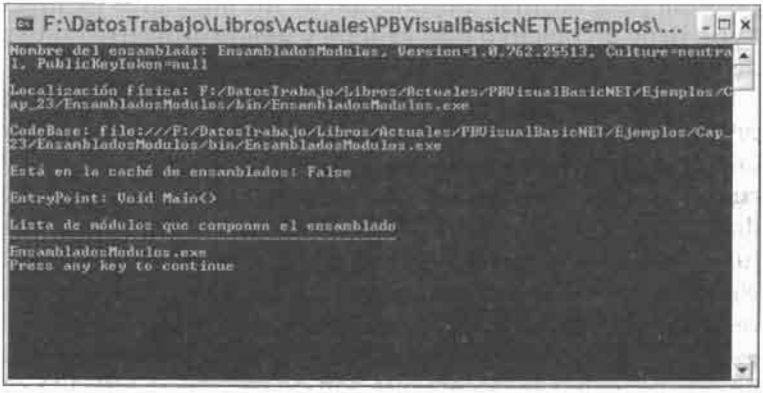


Figura 23.3. Mostramos la lista de módulos que componen el ensamblado

En este caso hemos utilizado el método `GetModules()` para enumerar los módulos del ensamblado. También podemos usar el valor devuelto como un arreglo, utilizando un índice numérico para acceder a cada uno de los módulos. El primero, que tiene por índice cero, es siempre el módulo que podríamos considerar principal, donde se encuentra el punto de entrada.

Tipos definidos en un ensamblado

Un ensamblado es una unidad lógica, formada por uno o más módulos físicos, en el que viven los tipos de datos públicos definidos en éstos. Son tipos las clases, estructuras, enumeraciones, delegados, etc. Al igual que podemos saber qué módulos forman un ensamblado, también es fácil conocer la lista de tipos que hay en su interior, así como conocer los detalles de cada tipo.

En este caso el método que nos interesa se llama `GetTypes()` y podemos encontrarlo tanto en la clase `Assembly` como en `Module`. En el primer caso obtendríamos una lista de los tipos existentes en el ensamblado, mientras que en el segundo la lista sería únicamente de los tipos de un cierto módulo. En nuestro caso, tomando como base el ejemplo anterior, el resultado sería el mismo dado que el ensamblado se compone de un solo módulo.

Lo que devuelve el método `GetTypes()` es un arreglo de objetos de la clase `Type`. Tal y como se indicó anteriormente, es posible obtener un objeto de esa clase con la información de cualquier variable u objeto mediante el método `GetType()` definido en la clase `Object`, ascendiente común de todos los tipos existentes en Visual Basic .NET.

La clase `Type`

A pesar de estar definida en el ámbito `System` y ser una clase fundamental para el funcionamiento de los diseñadores y editores de Visual Basic .NET, la clase `Type` está derivada no de `Object` sino de `MemberInfo`, una clase que sí se aloja en el ámbito `System.Reflection`. Mientras que `MemberInfo` está pensada para facilitar información de los miembros de una clase o estructura, `Type` tiene un objetivo más general y puede utilizarse tanto para obtener datos de una clase como de cualquiera de sus miembros.

Uno de los pocos miembros que `Type` hereda de `MemberInfo` es `Name`, una propiedad en la que encontraremos el nombre del tipo. Generalmente el nombre no nos dirá mucho a menos que conozcamos el tipo de antemano, caso en el cual no necesitaríamos los servicios de reflexión de la plataforma .NET. Mediante una treintena de métodos de la clase `Type`, todos ellos siguen la nomenclatura `IsXXXX()` y devuelven un valor de tipo `Boolean`, podremos saber si el elemento es una clase: `IsClass()`; una interfaz: `IsInterface()`; una enumeración: `IsEnum()`; tiene ámbito público: `IsPublic()`; o no: `IsNotPublic()`, es de uso final y no puede utilizarse como base: `IsSealed()`, etc. Tan sólo tiene que consultar la documentación de referencia sobre la clase `Type` para conocer todos los métodos disponibles.

Una vez determinada la naturaleza del tipo, podemos utilizar los métodos de la clase `Type` para recuperar una lista de sus constructores, miembros, propiedades, métodos, eventos, etc. También podemos comprobar si el tipo cuenta con un miembro en concreto. Los métodos que nos interesan, en este caso, son `GetMethods()`, `GetEvents()`, `GetConstructors()` y `GetProperties()`, para el primer supuesto, o `FindMembers()` en el segundo.

La información devuelta por todos esos métodos son objetos de clases como `MethodInfo`, `EventInfo`, `ConstructorInfo` y `PropertyInfo`, que se caracterizan por contar con elementos que nos permiten determinar, por ejemplo, el tipo de una propiedad, la lista de parámetros de un método. Todas esas clases están derivadas de `MemberInfo`, por lo que pueden tratarse de manera genérica determinando el tipo de cada miembro mediante la propiedad `MemberType` de `MemberInfo`. Ésta puede tener uno de los valores que se enumeran en la tabla 23.1. Con un simple condicional podríamos convertir al tipo adecuado y obtener información específica sobre él.

Tabla 23.1. Valores de la enumeración `MemberTypes`

Constante	El miembro es ...
`Constructor`	Un constructor
`Event`	Un evento
`Field`	Una variable
`Method`	Un método
`Property`	Una propiedad
`NestedType`	Clase, interfaz, estructura, etc.
`Custom`	Personalizado

En caso de que el valor de `MemberType` sea `NestedType` nos encontramos con la definición de un nuevo tipo. Es lo que ocurre, por ejemplo, cuando una clase alberga la definición de otra clase, supuesto que conocimos en los primeros capítulos.

Parámetros de un método

En caso de que tengamos información sobre un método, los métodos de acceso de una propiedad o un constructor, puede interesarnos conocer la lista de parámetros que necesita para ser invocado. La información de un método se devuelve en un objeto de clase `MethodInfo`, mientras que la de un constructor se entrega en un objeto `ConstructorInfo`. Ambos están derivados de la clase `MethodBase`, de tal forma que cuentan con muchos elementos en común.

Si lo que tenemos es una propiedad, podemos obtener el objeto `MethodInfo` correspondiente a su método de lectura mediante `GetGetMethod()` y el de escritura con `GetSetMethod()`. De esta forma, nos basta con conocer uno de los métodos de la clase `MethodBase`: `GetParameters()`. Éste devuelve un arreglo de objetos `ParameterInfo`, conteniendo cada uno de ellos el nombre, tipo y atributos de un parámetro.

Hay que tener en cuenta que todas las clases mencionadas cuentan con el método `GetType()` y, por tanto, podemos obtener su tipo y descubrir información acerca de ellas de forma dinámica. También es importante conocer las

relaciones entre algunas de estas clases para conocer las posibles conversiones. El tipo `MemberInfo`, por ejemplo, es genérico y puede contener una referencia a un `MethodInfo` o un `PropertyInfo`, pudiendo convertirse a dichos tipos mediante la función `CType()`.

Jerarquía de tipos de un ensamblado

Como resumen de lo visto hasta ahora sobre el descubrimiento de tipos en ejecución, información que deberá complementar con la referencia de Visual Studio .NET sobre las clases mencionadas, vamos a desarrollar un sencillo programa que muestre un árbol jerárquico de los tipos contenidos en un ensamblado, enumerando también sus miembros. Nuestro objetivo será obtener un resultado como el de la figura 23.4, que corresponde al programa ya en funcionamiento y mostrando el contenido del ensamblado `JerarquiaTipos`. Como se aprecia en esa figura, el ensamblado tiene tres tipos de primer nivel: `Dias`, `Agenda` y `Form1`, que son una enumeración, una estructura y una clase, respectivamente. Debajo podemos ver todos sus miembros y, entre paréntesis, la categoría a la que pertenece. En el caso de los métodos también se facilita la lista de parámetros.

El componente que ocupa todo el espacio disponible en la ventana es un `TreeView`, similar al `ListView` que conocimos en un capítulo previo que se caracteriza por contar con uno o más nodos de primer nivel que pueden contener a otros en su interior. Cada nodo se corresponde con un objeto `TreeNode`, clase que dispone de una propiedad, llamada `Nodes`, conteniendo la colección de nodos hijo. Existe un método `Add()` que se encarga de añadir nuevos nodos y devolver una referencia a ellos, de tal forma que es fácil construir un árbol como el de la figura 23.4.

Asumiendo que hemos iniciado una nueva aplicación basada en formularios Windows e insertado un control `TreeView` en su interior, modificando la propiedad `Dock` para que ocupe todo el espacio disponible, el código que deberíamos introducir sería el siguiente:

```
Imports System.Reflection

  [illegible]
Public Enum Dias
  Lunes
  Martes
  Miercoles
  Jueves
  Viernes
  Sabado
  Domingo
End Enum

  [illegible]

Public Structure Agenda
```

```
  Public Sub New(ByVal mIDe As Integer, _
      ByVal mNombre As String, ByVal mTelefono As String)
    ID = mIDe
    Nombre = mNombre
    Telefono = mTelefono
  End Sub

  Public ID As Integer
  Public Nombre As String
  Public Telefono As String
End Structure

Public Class Form1
  Inherits System.Windows.Forms.Form


  Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)

    Dim EsteEnsamblado As [Assembly] = _
       [Assembly].Load("JerarquiaTipos")

    Dim Nodo As TreeNode = _
       TreeView1.Nodes.Add(EsteEnsamblado.FullName)
    Dim Tipo As Type


    For Each Tipo In EsteEnsamblado.GetTypes
      EnumeraTipos(Nodo, Tipo)
    Next
  End Sub


  Private Sub EnumeraTipos(ByVal Nodo As TreeNode, _
                           ByVal Tipo As Type)

    Dim Hijo As TreeNode = Nodo.Nodes.Add(Tipo.Name & _
       " Inherits " & Tipo.BaseType.Name)
    EnumeraMiembros(Hijo, Tipo)
  End Sub


  Private Sub EnumeraMiembros(ByVal Nodo As TreeNode, _
                              ByVal Tipo As Type)
    Dim Info As MemberInfo
    Dim Hijo As TreeNode


    For Each Info In Tipo.GetMembers

      Hijo = Nodo.Nodes.Add(Info.Name & " (" & _
         Info.MemberType.ToString() & ")")

      If Info.MemberType = MemberTypes.Method Then

        EnumeraParametros(Hijo, Info)
      End If
    Next
  End Sub
```

```
  Private Sub EnumeraParametros(ByVal Nodo As TreeNode, _
                                ByVal InfoMetodo As MemberInfo)
    Dim Info As ParameterInfo


    For Each Info In CType(InfoMetodo, MethodInfo).GetParameters

      Nodo.Nodes.Add(Info.Name & " - " & _
          Info.ParameterType.ToString)
    Next
  End Sub
End Class
```

Figura 23.4. Vista jerárquica de tipos de un ensamblado

El código prácticamente está auto-documentado con los comentarios que se han introducido. Uno de los pocos puntos destacables es la llamada al método `EnumeraParametros()` desde `EnumeraMiembros()`. En éste último se dispone de un objeto de clase `MemberInfo` con información genérica sobre el tipo de miembro, objeto que se entrega al primero. En dicho método se efectúa una conversión de `MemberInfo` a `MethodInfo` para poder así recuperar la lista de parámetros. Esto es posible, lógicamente, porque la variable `Info` del método `EnumeraMiembros` aunque es de tipo `MemberInfo` apunta a un objeto `MethodInfo`, hecho que hemos comprobado mediante la propiedad `MemberType`. Si ésta no contuviese el valor `MemberTypes.Method` la citada conversión provocaría una excepción.

Uso dinámico de objetos

Si la información recuperada durante la ejecución tan sólo nos sirviera para mostrarla al usuario, lo cierto es que su utilidad sería mínima. Una aplicación que usa los servicios de reflexión para recuperar información generalmente la utiliza con otro fin, no para mostrarla al usuario, como puede ser el acceso dinámico a los objetos, sin conocer previamente su localización ni tipo.

Algunas de las clases citadas en los puntos previos, como `MethodInfo` y `PropertyInfo`, disponen de miembros que facilitan su invocación, recuperación o modificación de valor. `MethodInfo`, por ejemplo, cuenta con un método llamado `Invoke()` que efectúa una llamada al método representado sobre un objeto que debemos facilitar como parámetro. De manera análoga, `PropertyInfo` cuenta con los métodos `GetValue()` y `SetValue()` cuya finalidad es fácil suponer.

Hasta ahora, en los ejemplos desarrollados en capítulos previos, si hemos querido usar un cierto objeto, por ejemplo un formulario, éste ha sido definido en un módulo del mismo proyecto o, como mucho, se trataba de una clase alojada en una biblioteca de enlace dinámico cuya referencia hemos importado. En definitiva, el entorno conocía el tipo del objeto a crear y usar, de tal forma que permitía crearlo y acceder a sus métodos y propiedades sin ningún problema.

Las dificultades surgen cuando los objetos a usar no tienen tipos conocidos durante el desarrollo de la aplicación, por ejemplo porque se vayan a facilitar posteriormente en forma de bibliotecas fácilmente actualizables. ¿Cómo podemos crear un objeto de una clase sin disponer de una referencia a ella durante la fase de desarrollo? En ese caso usaríamos lo que se denomina *early binding* o enlace temprano o en fase de compilación. Como no existe esa posibilidad, tendremos que recurrir al *late binding* o enlace tardío o en fase de ejecución.

La clase `Activator`

El primer paso para poder utilizar una clase mediante enlace en fase de ejecución, creándola y accediendo a sus miembros de manera dinámica, es crear

un objeto a partir de esa clase. Ése es el cometido de los métodos compartidos de la clase `Activator`, que podemos usar tanto para crear objetos locales como remotos, así como para recuperar referencias a objetos que ya están ejecutándose en algún punto.

Los dos métodos que más nos interesan de esta clase son `CreateInstance()` y `CreateInstanceFrom()`. El primero de ellos crea un objeto de una clase a partir de su información de tipo, que deberemos facilitar mediante un objeto `Type`. Ya sabemos que podemos abrir un ensamblado y enumerar todos sus tipos, por lo que no tendríamos más que entregar esos tipos a `CreateInstance()` para crear objetos. El segundo método, `CreateInstanceFrom()`, necesita dos parámetros: el nombre de un ensamblado y el nombre de la clase que vamos a usar para crear el objeto. En un solo paso se abre el ensamblado, localiza el tipo y crea el objeto, sin necesidad de pasos previos.

Asumiendo que hemos creado una biblioteca de clases y alojado en ella una clase llamada `Formulario` en el ámbito `Formulario`, podríamos usar el código siguiente, en otro proyecto alojado en la misma carpeta, para abrir el ensamblado, recuperar una referencia al tipo y crear el objeto.

```
Dim Ensamblado As [Assembly] = _
  [Assembly].LoadFrom("..\..\Formulario\bin\Formulario.dll")

Dim Clase As Type = _
  Ensamblado.GetType("Formulario.Formulario")

Dim Objeto As Object = _
  Activator.CreateInstance(Clase)
```

Observe que el tipo de la variable `Objeto` es `Object` y no `Formulario`. No podríamos utilizar el tipo `Formulario` como tal ya que en el proyecto no existe ninguna referencia a él, por eso utilizamos una referencia genérica a un objeto, sin conocer nada más de él.

Nota

Para poder probar el código anterior, y el del punto siguiente, cree una biblioteca de clases y defina en ella una clase llamada `Formulario` derivada de `Form` y que cuente con un solo método, llamado `Saludos()`, que se limite a mostrar un mensaje.

Invocación dinámica

Disponiendo de una referencia a nuestro objeto en la variable `Objeto`, podríamos invocar al método `Saludos()` o, ya que `Formulario` es una clase derivada de `Form`, modificar cualquiera de las propiedades e invocar a cualquier método heredado. No podemos, sin embargo, escribir sentencias tales

como `Objeto.Visible = True`, ya que `Objeto` es de tipo `Object` y en dicho tipo no existe la propiedad `Visible`.

La variable `Clase` definida en el anterior fragmento de código, conteniendo la información de tipo de nuestro objeto, puede ser utilizada para obtener datos sobre los métodos y propiedades de la clase, tan sólo tendríamos que usar `GetMethod()` y `GetProperty()`, en este caso, para obtener un objeto `MethodInfo` o `PropertyInfo` con información sobre el método o propiedad que nos interese.

En el código siguiente se utiliza esta técnica para invocar a los métodos `Saludos()` y `Show()`, así como para modificar el valor de la propiedad `Text`. Al ejecutar el código aparecerá primero un mensaje y, a continuación, el formulario con el título modificado.

```
Dim Metodo As MethodInfo = Clase.GetMethod("Saludos")

Metodo.Invoke(Objeto, Nothing)

Dim Propiedad As PropertyInfo = Clase.GetProperty("Text")

Propiedad.SetValue(Objeto, "Nuevo título de la ventana", Nothing)

Metodo = Clase.GetMethod("Show")

Metodo.Invoke(Objeto, Nothing)
```

En este ejemplo hemos asumido que el ensamblado `Formulario.dll` contenía un tipo `Formulario` en el ámbito `Formulario` y que dicho tipo, una clase, dispone de ciertos métodos y propiedades. En la práctica, sin embargo, podemos usar los métodos explicados anteriormente para conocer los tipos que hay en el ensamblado, identificar los métodos y propiedades, así como sus tipos, e invocarlos realmente sin conocerlos de manera previa. No obstante, sólo con lo que hemos hecho tenemos un programa que muestra un formulario alojado en una biblioteca de enlace dinámico sin contar con un enlace en la fase de diseño. Esto nos permite modificar esa biblioteca siempre que nos interese, por ejemplo modificando el diseño del formulario o la definición del método `Saludos()`, sin por ello tener que recompilar la aplicación principal. Bastaría con redistribuir la biblioteca para actualizar el programa, siempre que se conserve el nombre del tipo.

Nota

En lugar de facilitar al método `LoadFrom()` de la clase `Assembly` una constante, con el camino y nombre de la biblioteca, también existe la posibilidad de permitir seleccionar al usuario la biblioteca o que ésta venga determinada por algún parámetro de configuración.

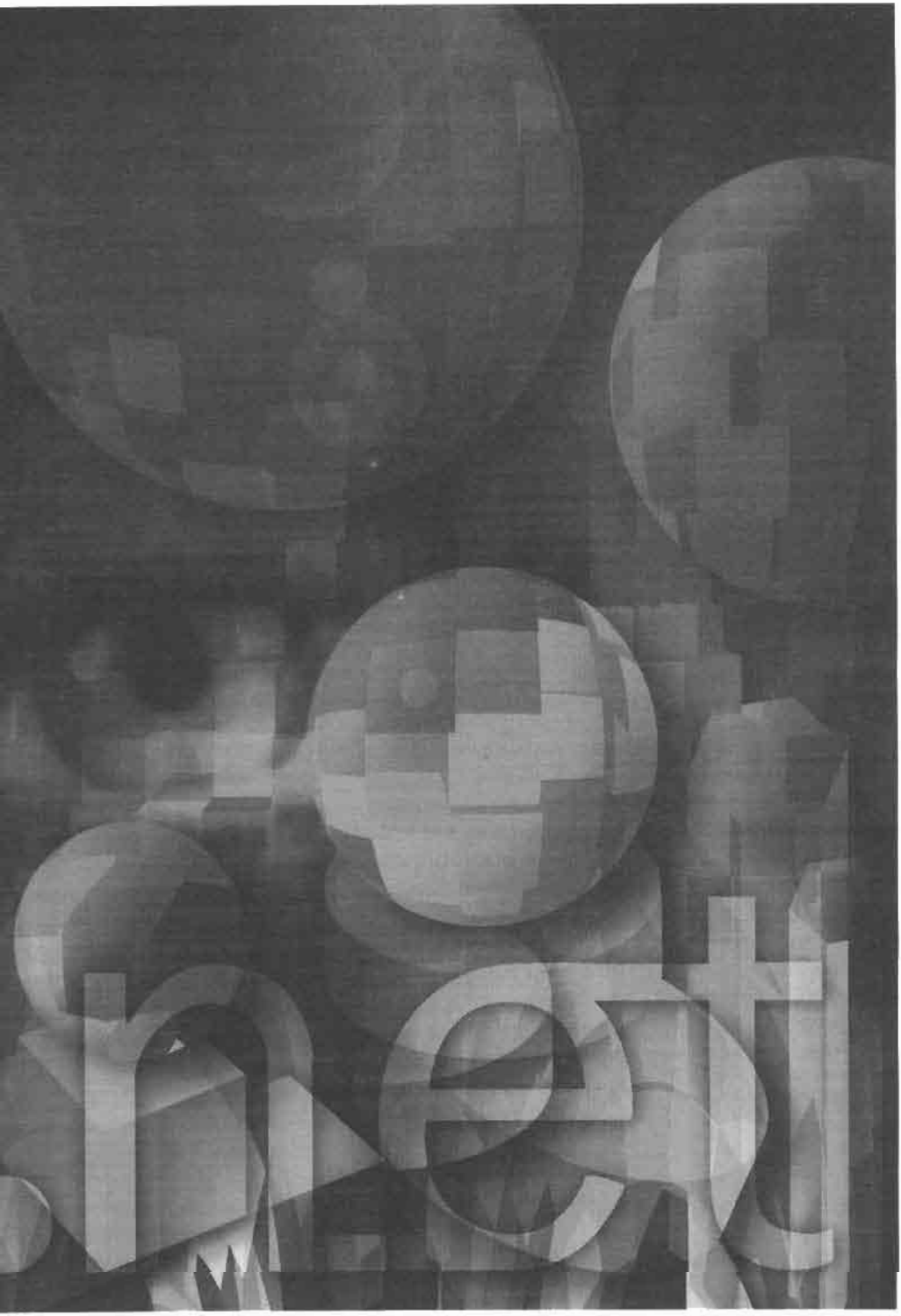
Puntos clave

- La plataforma .NET dispone de una serie de servicios que hacen posible el descubrimiento de la información de tipos durante la ejecución, facilitando técnicas como el uso dinámico de objetos sin conocimiento previo en la fase de diseño.
- Las clases `Assembly` y `Module` identifican a un ensamblado y un módulo, respectivamente, siendo la primera la más interesante al facilitar información de todos sus tipos.
- Para aprovechar los servicios de reflexión es fundamental conocer la clase `Type`, base de todo el mecanismo de recuperación de información de tipos.
- Mediante los métodos de `Type` es posible crear una jerarquía de elementos conociendo los tipos y miembros de todos los objetos.
- Podemos crear objetos de clases sin contar con una referencia en fase de diseño sirviéndonos de la clase `Activator`.
- Los objetos de las clases `MethodInfo` y `PropertyInfo` facilitan la invocación, lectura y modificación dinámicos.

Resumen

Mediante las técnicas que hemos conocido en este capítulo podemos conocer en ejecución objetos y elementos que en fase de diseño no se conocían, obteniendo información sobre clases, enumeraciones, estructuras, métodos, propiedades, eventos y, en general, todos los componentes de un ensamblado. Estos servicios son los que usan los propios diseñadores de Visual Basic .NET para, por ejemplo, facilitar una lista de miembros cuando en el editor se introduce el nombre de un objeto o enumerar las propiedades en la ventana Propiedades.

Lo más interesante de estos servicios, a menos que vayamos a crear algún diseñador o herramienta similar, es que nos permiten usar objetos de forma dinámica, sin conocerlos de antemano, aportando mucha flexibilidad al diseño de una aplicación. Hemos visto cómo es posible usar un elemento alojado en una biblioteca, biblioteca que podría actualizarse siempre que fuese necesario sin afectar a la aplicación principal.



24

Interactuación con COM y el API de Windows

Visual Basic .NET es un lenguaje que, aunque desarrollado sobre la base de Visual Basic 6, está diseñado para operar sobre una plataforma totalmente nueva, la plataforma .NET, con servicios mucho más avanzados y simples que los encontrábamos en versiones previas. Hay que tener en cuenta, no obstante, que en la actualidad son miles los componentes COM usados en miles de aplicaciones, así como que muchas de ellas tienen necesidad de acceder a los servicios del propio sistema, lo que se conoce habitualmente como API de Windows.

Que la plataforma .NET ofrezca una alternativa mucho más potente, flexible y simple no implica, sin embargo, una ruptura total con el pasado. En la plataforma .NET existen unos servicios, denominados de interacción y alojados en el ámbito `System.Interop`, que hacen posible la comunicación bidireccional entre .NET y COM. Desde Visual Basic .NET podemos usar componentes COM, así como ofrecer nuestros componentes .NET a herramientas que trabajan con COM.

También el acceso al API de Windows es totalmente factible, sin bien no significa que sea algo recomendable. Siempre que en la plataforma .NET exista un servicio para hacer lo que necesitamos es preferible usarlo en detrimento del API de Windows, consiguiendo así una mayor independencia del sistema. Aunque en este momento sea un tema que no nos afecte demasiado, en el futuro

podríamos necesitar que nuestras aplicaciones operasen sobre la plataforma .NET en otros sistemas operativos para los que esté disponible, y si tenemos llamadas al API de Windows lógicamente tendremos problemas para conseguir ese objetivo.

En este breve capítulo abordaremos tres temas distintos: cómo utilizar componentes COM o controles ActiveX desde un proyecto Visual Basic .NET, cómo exportar un componente .NET para que pueda ser usado mediante COM y, finalmente, cómo acceder al API de Windows.

Uso de componentes COM

Posiblemente tengamos en uso en nuestros proyectos ya desarrollados controles ActiveX o componentes COM de los que, de momento, no podamos prescindir. Para trabajar en Visual Studio .NET, por tanto, precisaríamos esos objetos en nuevos proyectos o durante la actualización/conversión de los ya desarrollados.

Dependiendo de que el elemento en cuestión sea un control ActiveX, que tomamos habitualmente desde la Caja de herramientas de Visual Basic, o un componente COM simple, que no aparece en esa ventana, el proceso de importación será uno u otro.

Lo habitual es conocer de antemano el componente COM que necesitamos utilizar, de tal manera que sea posible agregar una referencia durante la fase de diseño generando el correspondiente RCW. También podemos, no obstante, acceder a estos componentes de manera dinámica.

Importación de controles ActiveX

Comencemos por el supuesto más simple que, en este caso, es que dispongamos de un control ActiveX y que necesitemos utilizarlo en un proyecto de Visual Basic .NET. Dicho control deberá estar registrado en el sistema, acción que se efectúa normalmente durante la instalación del componente o el producto del que forme parte. Lo único que tenemos que hacer es utilizar la opción Personalizar cuadro de herramientas del Cuadro de herramientas, seleccionando de la página **Componentes COM** (véase figura 24.1) el control que deseamos usar. Si no aparece en la lista siempre podemos pulsar el botón **Examinar** para localizar la biblioteca donde se encuentra dicho control.

Nota

Si dispone de una biblioteca en la que se aloja el control a utilizar y éste no se encuentra registrado en el sistema, puede efectuar el registro introduciendo en la consola del sistema el comando `regsvr32` seguido del nombre de la biblioteca incluyendo la extensión.

Figura 24.1. Importamos el control de hoja de cálculo de Microsoft Office

Al importar un control ActiveX, Visual Basic .NET se encarga de generar un envoltorio o *proxy*, también conocido como RCW (*Runtime Callable Wrapper*). Éste actúa como intermediario entre COM y .NET ocupándose del proceso conocido como *marshalling*, consistente, básicamente, en la conversión de tipos entre un modelo y otro de componentes. El RCW se ocupa también de la adaptación entre COM y el CLR, de tal forma que el cliente Visual Basic .NET cree que está utilizando un control .NET, mientras que éste piensa que está funcionando en un cliente COM.

Tras la importación del componente no tiene más que utilizarlo como usaría cualquier otro, insertándolo en el formulario, personalizando sus propiedades y respondiendo a sus eventos. En la figura 24.2 puede ver el componente de hoja de cálculo de Microsoft Office, un control ActiveX, en un formulario de Visual Basic .NET.

Importación de librerías de tipos

El segundo supuesto, de los mencionados al principio, es que necesitemos usar componentes COM que no necesariamente sean controles. Para ello tendremos que importar la librería de tipos generando el RCW que, en este caso, deberemos usar para crear el componente COM. Suponga que deseamos utilizar desde Visual Basic .NET los servicios de DirectX 8, el API multimedia de Windows. El primer paso sería seleccionar la opción **Añadir referencia** de la carpeta **Referencias** en el **Explorador de soluciones**, abriendo la página **COM** en la ventana que aparece. Seleccionamos la librería de tipos, en este caso **DirectX 8 for Visual Basic Type Library**, tal y como se ha hecho en la figura 24.3. Al pulsar el botón **Aceptar** aparecerá un mensaje (véase figura 24.4) indicando que es necesario generar un ensamblado con el RCW para poder acceder al componente COM.

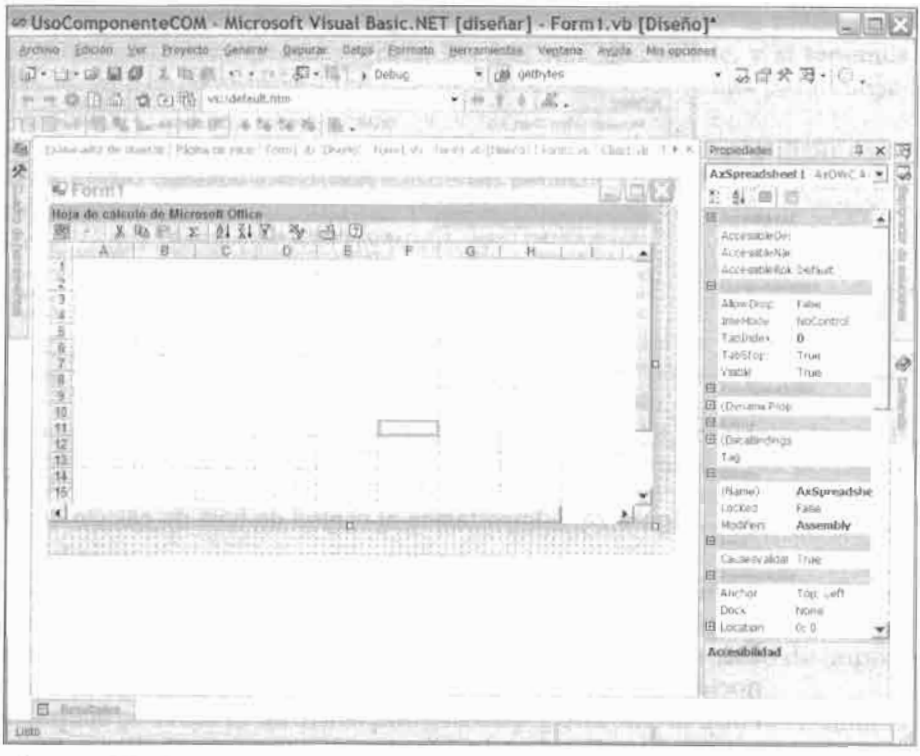


Figura 24.2. Aspecto del control ActiveX en el interior del formulario

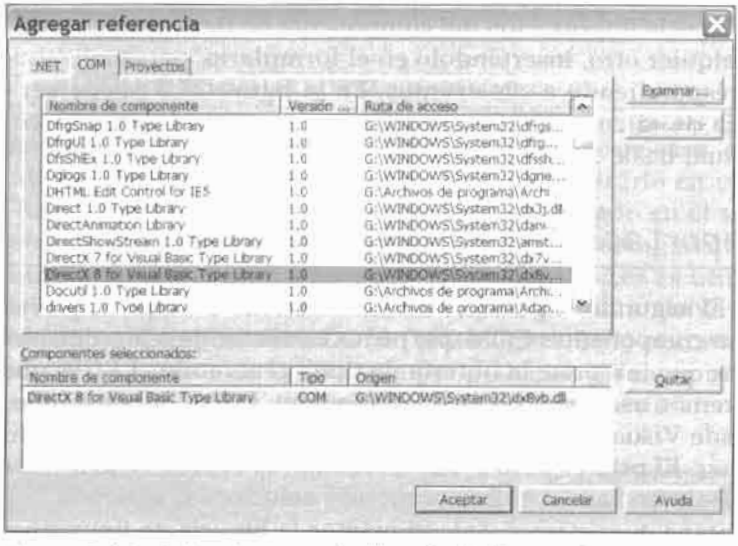


Figura 24.3. Seleccionamos la librería de tipos a importar

En la lista de referencias del proyecto debe aparecer el elemento `DxVBLibA`, el ámbito generado en el que se encuentran todos los elementos importados de

la librería de tipos. Puede usar el Examinador de objetos de Visual Basic .NET para ver el contenido de ese ámbito: estructuras, enumeraciones, clases, etc. Asumiendo que hemos importado la citada librería de tipos, introduciendo el código siguiente podríamos enumerar los dispositivos de salida de audio existentes en nuestro sistema:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
          ByVal e As System.EventArgs) Handles MyBase.Load

  Dim MyDX As New DxVBLibA.DirectX8()

  Dim MisDispositivos As DxVBLibA.DirectSoundEnum8 = _
              MyDX.GetDSEnum()
  Dim Indice As Integer, Resultado As String

  For Indice = 1 To MisDispositivos.GetCount
    Resultado = Resultado & _
        MisDispositivos.GetDescription(Indice) & vbCr
  Next
  MessageBox.Show(Resultado)
End Sub
```

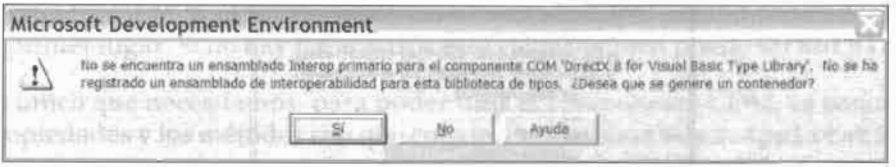


Figura 24.4. Visual Basic .NET nos avisa de la necesidad de generar el RCW para poder usar el componente

Dependiendo del ordenador en el que ejecute el programa el resultado será uno u otro. En la figura 24.5 puede ver el obtenido en mi propio sistema.

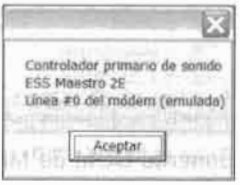


Figura 24.5. Lista de los controladores de audio obtenida mediante DirectX

Uso dinámico de componentes COM

En el capítulo previo vimos cómo podía utilizar la información de tipos obtenida en ejecución para usar objetos dinámicamente, sin conocerlos durante la fase de desarrollo. Tampoco es absolutamente necesaria importar una librería de tipos, generando el RCW, para poder utilizar un componente COM. Si sabemos cuál es su cadena de identificación, lo que se conoce habitualmente como

progid, podemos usar las clases `Type` y `Activator` para utilizar el componente de forma dinámica.

Para utilizar el método `CreateInstance()` de la clase `Activator` antes necesitamos información sobre el tipo de objeto que vamos a crear, información que en el capítulo anterior obteníamos mediante el método `GetType()` o enumerando los tipos de un ensamblado. En este caso, la información sobre el tipo está almacenada en el registro de Windows asociada a un CLSID (*Class Identifier*) y un *ProgId* o cadena de identificación. Los CLSID son números de 128 bits bastante difíciles de recordar, pero la cadena de identificación es mucho más descriptiva. En la figura 24.6 puede ver en la parte izquierda una serie de carpetas, correspondiente cada una de ellas a un CLSID de un componente. Una de ellas está abierta, apareciendo a la derecha su *ProgId*.

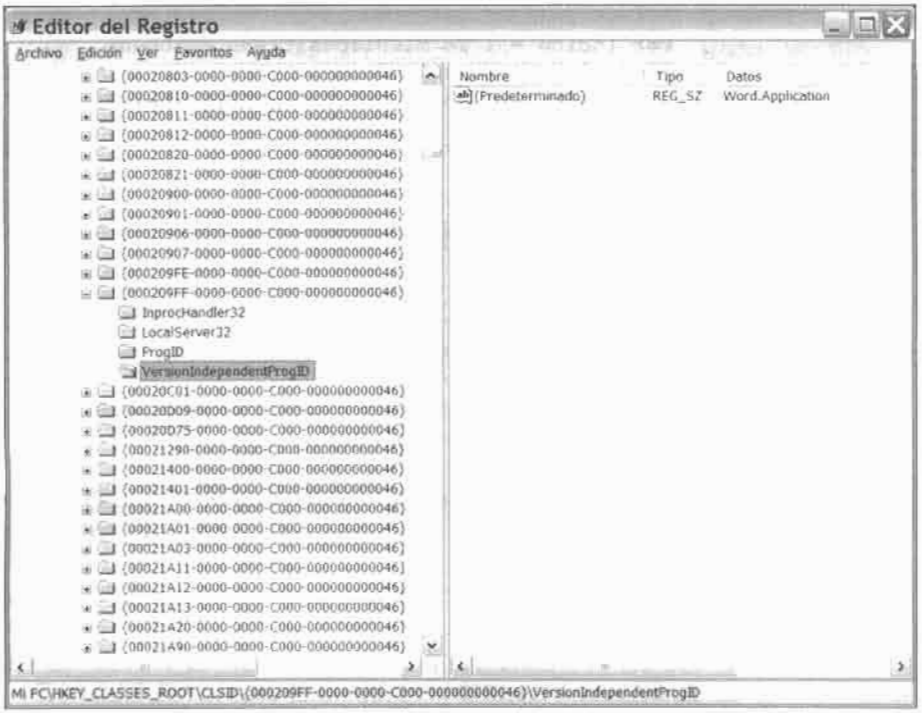


Figura 24.6. CLSID y *ProgId* de un componente COM de Microsoft Office

Para obtener un objeto `Type` con información de tipo a partir de una cadena de identificación, que en este caso es lo más fácil de recordar, usaremos el método `GetTypeFromProgID()` de la propia clase `Type`. Se trata de un método compartido, por lo que podemos llamarlo directamente sin más que poner delante el nombre de la clase. También tenemos a nuestra disposición el método `GetTypeFromCLSID()`, en caso de que vayamos a introducir el CLSID en lugar del *ProgID*.

Disponiendo de la información de tipo, utilizaríamos el método `CreateInstance()` de la clase `Activator` mencionado antes, creando el componen-

te y obteniendo una referencia que alojaríamos en una variable de tipo `Object`. A partir de aquí nos serviríamos reiteradamente del método `InvokeMember()` de la clase `Type` para leer o escribir en las propiedades del objeto e invocar a sus métodos.

Los parámetros que debemos facilitar a dicho método son cinco:

- Una cadena de caracteres con el nombre de la propiedad o método a invocar.
- Una constante o combinación de constantes que permitan localizar al miembro cuyo nombre se entrega en primer lugar. Los más habituales son `BindingFlags.InvokeMethod`, `BindingFlags.GetProperty` y `BindingFlags.SetProperty` para llamar a un método, leer y modificar una propiedad, respectivamente.
- Un objeto `Binder` que facilita la limitación de la selección, por ejemplo para invocar a un método sobrecargado. Puede ser `Nothing`.
- La referencia al objeto sobre el cual quiere actuarse, que en nuestro caso sería el creado previamente con `Activator.CreateInstance()`.
- En último lugar debemos entregar un arreglo de tipo `Object` con los parámetros que sean necesarios para el método o propiedad indicados en primer lugar. Si no hay parámetros este valor también puede ser `Nothing`.

Lo único que necesitamos, para poder usar el componente COM, es conocer las propiedades y los métodos con que cuenta. Para utilizar `Word.Application` desde nuestro programa, por ejemplo, tendríamos que disponer de documentación sobre Word, sus métodos y propiedades, información que es fácil encontrar en la web de MSDN.

Puede realizar una sencilla prueba insertando en un formulario un botón y asociando el código siguiente al evento de pulsación:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click

  Dim Tipo As Type = Type.GetTypeFromProgID("Word.Application")

  Dim Objeto As Object = Activator.CreateInstance(Tipo)

  Tipo.InvokeMember("Visible", BindingFlags.SetProperty, _
                    Nothing, Objeto, New Object() {True})

  Dim Documentos As Object = Tipo.InvokeMember("Documents", _
          BindingFlags.GetProperty, Nothing, Objeto, Nothing)

  Dim Documento As Object = Documentos.GetType.InvokeMember( _
          "Add", BindingFlags.InvokeMethod, Nothing, _
          Documentos, Nothing)
```

```
    Dim Contenido As Object = Documento.GetType.InvokeMember( _
            "Content", BindingFlags.GetProperty, Nothing, _
            Documento, Nothing)

    Contenido.GetType.InvokeMember("InsertAfter", _
            BindingFlags.InvokeMethod, Nothing, Contenido, _
       New Object() {"Texto introducido desde Visual Basic .NET"})

    Documento.GetType.InvokeMember("SaveAs", _
          BindingFlags.InvokeMethod, Nothing, Documento, Nothing)
  End Sub
```

Tras crear el componente, modificamos el valor de la propiedad `Visible` para que la ventana de Word aparezca en pantalla. A continuación recuperamos la propiedad `Documents`, que es una colección, para añadir un nuevo documento. Éste se compone de diversas partes, entre ellas un contenido que está representado por la propiedad `Content`. Nos servimos de ella para insertar un texto y, finalmente, guardamos el documento. Al ejecutar el programa verá que se abre Word, lógicamente siempre que lo tenga instalado en su sistema, y que se genera un nuevo documento con un texto y se guarda automáticamente.

Exposición de componentes .NET a COM

Cuando aparece una nueva herramienta de desarrollo, como es en este caso Visual Basic .NET, lo habitual es que encontremos la necesidad de utilizar con ella componentes que ya teníamos de la versión anterior. Hemos visto en los puntos anteriores que podemos hacerlo sin ningún problema. El caso inverso, exponer componentes de la nueva herramienta para poder usarlos desde la antigua, es poco habitual, pero la plataforma .NET también la contempla.

La finalidad del RCW, como hemos visto, es servir de intermediario entre COM y .NET. Desde Visual Basic .NET utilizamos el componente como si fuese cualquier otro, por ejemplo creándolo e invocando a sus métodos. Oculto queda el trabajo del RCW que, para que esas llamadas sean posibles, tiene que convertirlas y entendérselas con las clásicas interfaces de COM, como `IUnknown` e `IDispatch`.

Para que un componente .NET pueda ser usado desde COM el caso es el inverso, el uso de interfaces COM debe traducirse apropiadamente, teniendo en cuenta que la herramienta COM ejecuta código nativo Win32 y que el componente .NET es código MSIL. Lo que necesitamos, para hacer posible esa comunicación, es generar un CCW (*COM Callable Wrapper*).

Además hay que tener en cuenta el sistema que usa COM para localizar los componentes. Éstos deben estar registrados apropiadamente, introduciendo su CLSID, *ProgID*, localización física y otros datos en el registro de configuraciones de Windows. Por ello, un ensamblado .NET que contiene un componente que va a exponerse a COM debe ser un ensamblado compartido, para lo cual debe contar con una identificación única.

Nota

En COM no existen los ámbitos con nombre, que es el sistema utilizado en .NET para identificar de forma inequívoca a los diversos elementos que pueden componer un proyecto. Por ello es necesario utilizar otro sistema de identificación.

Desarrollo y opciones del componente

Vamos ahora a diseñar un componente .NET alojándolo en una biblioteca de clases, como es habitual. Este componente estará derivado de `System.Windows.Forms.Form` y sólo añadiremos un nuevo método, llamado `Saludos()`, que mostrará una ventana con un mensaje. Es, básicamente, el mismo componente que habíamos creado al final del capítulo anterior y que después usábamos de manera dinámica.

Más que el componente en sí, lo importante en este caso son las opciones de configuración necesarias para que pueda ser utilizado desde COM. Esas opciones serán siempre las mismas, independientemente de la finalidad concreta que tenga el componente. El primer paso será abrir la ventana de propiedades del proyecto, accediendo a la página **Nombre seguro** que aparece en la figura 24.7. En ella activamos la opción **Generar nombre seguro usando** y, a continuación, pulsamos el botón **Generar clave**. Con esto conseguimos que se cree una clase única para nuestro ensamblado, clave que permitirá la identificación inequívoca de manera similar a un CLSID de COM.

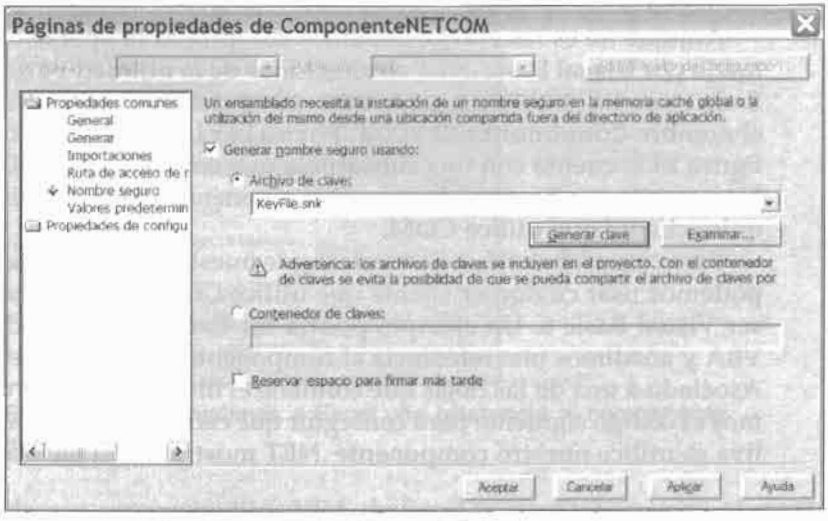


Figura 24.7. Generamos una clave para asociarla al ensamblado

Usando esa clave, al compilar el proyecto queremos que el ensamblado se registre adecuadamente para que COM pueda encontrarlo. Con este fin tenemos

que activar la opción Registrar para interoperabilidad COM de la página Generar en la misma ventana de opciones de proyecto (véase figura 24.8). Podemos cerrar la ventana de propiedades pulsando el botón **Aceptar** y compilar el proyecto. En la ventana de resultados veremos una indicación acerca del registro en COM.

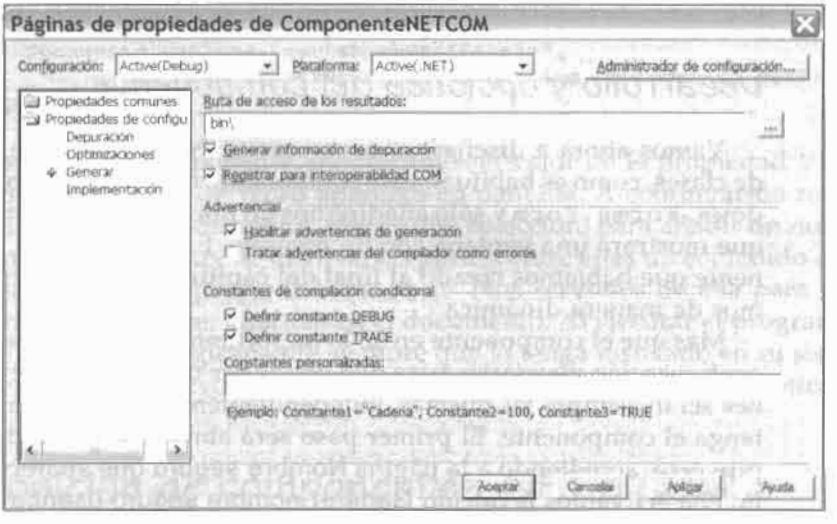


Figura 24.8. Activamos la opción de registro en COM

Uso del componente desde un cliente COM

Aunque no es necesario, podemos comprobar la operación de registro efectuada por Visual Basic .NET sirviéndonos de la utilidad `regedit` de Windows. En la rama `Mi PC\HKEY_CLASSES_ROOT` debemos encontrar una entrada con el nombre `ComponenteNETCOM.FormularioNET` que, como se aprecia en la figura 24.9, cuenta con una subcarpeta que contiene el CLSID del componente. No necesitamos hacer nada más, el componente ya puede utilizarse desde cualquier cliente que utilice COM.

Para comprobar el funcionamiento de nuestro componente .NET desde COM podemos usar cualquier cliente que utilice COM, no tiene necesariamente que ser Visual Basic 6. Un ejemplo podría ser Excel 2000. Accedemos al editor de VBA y añadimos una referencia al componente, como se hace en la figura 24.10. Asociado a una de las hojas que contiene el libro abierto por defecto, introducimos el código siguiente para conseguir que cada vez que se cambie la celdilla activa se utilice nuestro componente .NET mostrando el mensaje y el formulario.

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
   Dim MiComponente As New ComponenteNETCOM.FormularioNET
   MiComponente.Saludos
   MiComponente.Text = "Formulario .NET en Excel 2000"
   MiComponente.Show
End Sub
```

Figura 24.9. Información introducida por Visual Basic .NET en el registro de Windows

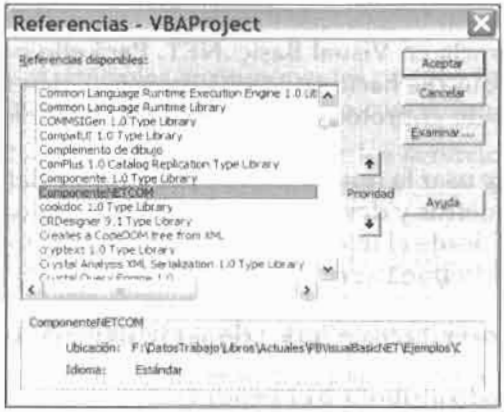


Figura 24.10. Añadimos a Excel una referencia al componente antes registrado

Podemos cerrar el editor VBA de Excel y comprobar cómo al movernos de una celdilla a otra aparece la ventana con el mensaje y a continuación el formulario, un formulario diseñado con Visual Basic .NET y que estamos utilizando desde una aplicación, en este caso Excel, que no funciona sobre la plataforma .NET sino nativamente en Windows.

Tal y como puede verse, la interoperabilidad es total y simple para el desarrollador.

Acceso al API de Windows

La plataforma .NET dispone de tantos servicios que difícilmente encontraremos una razón para recurrir directamente al API de Windows, esas excepciones, no obstante, existen, y vamos a ver cómo podemos actuar cuando nos encontremos con ellas.

Para utilizar una función del API de Windows es fundamental conocer su nombre, localización, tipo de dato de retorno y la lista de parámetros. Toda esa información la encontraremos en la documentación conocida como *Platform SDK* de Microsoft, accesible desde la sede web de MSDN. Intentar la declaración y llamada a una función sin estar seguros, por ejemplo, de los tipos de los parámetros es una práctica peligrosa.

Lógicamente, hay que conocer la correspondencia entre tipos de datos Win32 y los tipos de Visual Basic .NET. Si una función Win32 devuelve un valor de tipo `LONG`, por ejemplo, debemos saber si en Visual Basic .NET lo queremos tratar como un número, caso en el que utilizaríamos el tipo `Integer`, o bien como un puntero, recurriendo entonces al tipo `IntPtr`.

Declaración de la función

El primer paso que tendremos que dar para acceder a una función del API de Windows es declararla en Visual Basic .NET. Para ello podemos utilizar la sentencia `Declare`, como se hacía en Visual Basic 6, o bien utilizar el atributo `DllImport` precediendo el prototipo de la función, indicando la biblioteca en que se encuentra.

Suponga que quiere usar la función `GetTickCount()` del API de Windows. Esta no necesita parámetros y devuelve un entero indicando el número de *ticks* de reloj transcurridos desde el inicio de Windows. Podríamos hacer la declaración usando la sentencia `Declare`:

```
Declare Function GetTickCount Lib "kernel32.dll" () As Integer
```

O bien el mencionado atributo `DllImport`:

```
<DllImport("kernel32.dll")> _
Public Shared Function GetTickCount() As Integer

End Function
```

Independientemente del método usado, tenemos en el proyecto un identificador, en este caso `GetTickCount()`, que hace referencia a una función Win32 alojada en una biblioteca clásica con código nativo.

Nota

Para poder usar el atributo `DllImport` tendrá que añadir al proyecto una sentencia `Imports` para importar el contenido del ámbito `System.Runtime.InteropServices`. También puede crear una referencia completa: `System.Runtime.InteropServices.DllImport`, prescindiendo de la importación.

Uso de las funciones

Efectuada la declaración, las funciones del API de Windows las usaremos desde nuestro programa Visual Basic .NET como si de cualquier otra función se tratase. En realidad aparecen como métodos compartidos, no asociados a ningún objeto, y sabemos perfectamente cómo utilizar esos métodos. Contando con la declaración efectuada en el punto anterior, podemos disponer en un formulario un botón y una etiqueta de texto mostrando en la propiedad `Text` de ésta el valor devuelto por `GetTickCount`. Cada vez que pulse el botón verá que el número de pulsos aumenta, como es de esperar.

Puntos clave

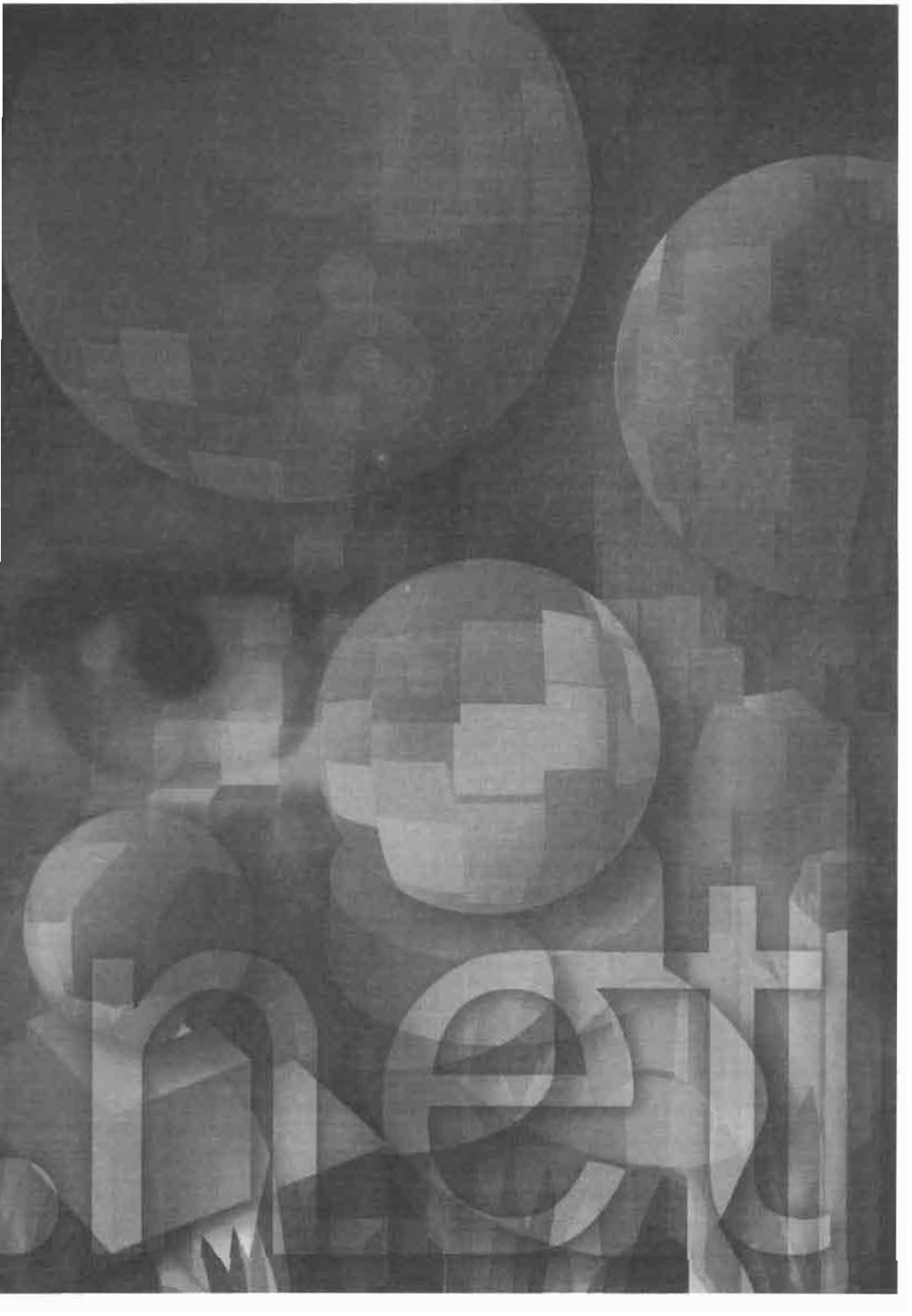
- Desde un proyecto Visual Basic .NET podemos utilizar cualquier componente COM generando un RCW (*Runtime Callable Wrapper*), un intermediario que efectúa todo el trabajo de adaptación entre .NET y COM.
- Los controles y componentes pueden importarse desde un proyecto y ser usados como cualquier otro componente .NET gracias al RCW.
- También pueden utilizarse los servicios de descubrimiento de tipos en ejecución para operar dinámicamente sobre cualquier componente COM automatizable.
- Un componente .NET puede configurarse para generar un CCW (*COM Callable Wrapper*) e introducir en el registro de Windows la información necesaria para poder ser usado desde clientes COM.
- Mediante el atributo `DllImport`, o la sentencia `Declare`, podemos acceder a cualquier servicio del API de Windows.

Resumen

Visual Basic .NET es un producto que acaba de presentarse, a principios de 2002, y llega a un campo en el que las aplicaciones y componentes nativos para Windows son el estándar actual. Por ello es fundamental contar con servicios

que permitan interaccionar en ambos sentidos .NET -> COM y COM -> .NET, servicios que hemos conocido básicamente en este capítulo generando los RCW y CCW utilizando tan sólo las opciones del entorno de Visual Basic .NET. También podríamos recurrir a herramientas de la propia plataforma .NET, como `regasm` o `tlbimp`, que no son necesarias si disponemos de Visual Basic .NET.

También se ha explicado cómo acceder a las funciones nativas del API de Windows, sin entrar en detalles sobre las propias funciones ya que, en la práctica, debería evitarse su uso a favor de los servicios de la propia plataforma .NET. Utilizar funciones Win32 implica unas dependencias que tendrían que superarse lo antes posible.



25

Conversión de proyectos Visual Basic 6 a Visual Basic .NET

En todos los capítulos de este libro en los que se han propuesto ejemplos, la mayoría de ellos, siempre hemos partido de la creación de un nuevo proyecto utilizando alguno de los asistentes de Visual Basic .NET. La creación de nuevos proyectos, usando los servicios de la plataforma .NET y la nueva sintaxis del lenguaje, no plantea ninguna dificultad añadida a la del propio aprendizaje. Esta opción, la de partir de cero, es la apropiada para desarrollar nuevos proyectos, pero obviamente no la mejor si de lo que se trata es de evolucionar de un proyecto ya existente, escrito con Visual Basic 6, para convertirlo a Visual Basic .NET.

Visual Basic .NET dispone de un asistente capaz de convertir proyectos de la versión anterior, si bien no siempre sin problemas. Hay que tener en cuenta que Visual Basic .NET no dispone exactamente del mismo conjunto de controles que Visual Basic 6 y que, aparte de novedades, hay elementos del lenguaje que han desaparecido, como el tipo `Variant` o la declaración de arreglos indicando los índices inferior y superior.

Nuestro objetivo en este capítulo no es ni mucho menos el de describir toda la problemática con que podemos encontrarnos al convertir un proyecto de Visual Basic 6 a Visual Basic .NET, objetivo casi inalcanzable por la cantidad de situaciones distintas y específicas de cada tipo de proyecto. En su lugar vamos a ver un caso práctico, convirtiendo un proyecto simple y conociendo algunos de los elementos de compatibilidad introducidos en Visual Basic .NET.

El asistente para actualización

Siempre que abramos desde Visual Basic .NET un proyecto desarrollado con una versión previa de Visual Basic se pondrá automáticamente en marcha el asistente para actualización de Visual Basic, cuya ventana de bienvenida aparece en la figura 25.1. Para trabajar sobre un ejemplo concreto, tomaremos un proyecto del título *Guía práctica de Visual Basic 6.0*, del mismo autor y editorial, cuyos ejemplos puede encontrar en un paquete comprimido en la sección Complementos de `http://www.AnayaMultimedia.es`. Concretamente tomaremos un ejemplo del sexto capítulo cuya finalidad es facilitar la navegación por el sistema de archivos mostrando gráficos.

Figura 25.1. En la ventana de bienvenida se indican los pasos que seguirá el asistente para convertir nuestro proyecto

En el ejemplo mencionado se utilizan controles de Visual Basic 6.0, ningún componente ActiveX externo. Tampoco se recurre a ninguna función del API de Windows, tan sólo sentencias del propio lenguaje, nada más.

Nota

En el CD-ROM que acompaña al libro encontrará el proyecto original Visual Basic 6 correspondiente al proyecto que vamos a convertir.

Inicio del proceso

Para poner en marcha el asistente no tenemos más que abrir el archivo vbp del proyecto Visual Basic que va a actualizarse. Aparecerá la ventana mostrada en la figura 25.2 y la pulsación del botón **Siguiente** nos llevará a una serie de ventanas con distintas opciones. En este caso, dado que el proyecto es un programa ejecutable, las opciones disponibles no son muchas. Tendremos que indicar un nuevo camino donde se creará el proyecto Visual Basic .NET tomando como base el de Visual Basic 6.

El proceso de conversión será, generalmente, breve y mientras dura podremos ir viendo los módulos sobre los que está actuando el asistente (véase figura 25.2). Finalizada la conversión veremos en el Explorador de soluciones que tenemos un nuevo proyecto con exactamente los mismos elementos que había en el original, ya que en este caso concreto no se ha encontrado ningún problema irresoluble.

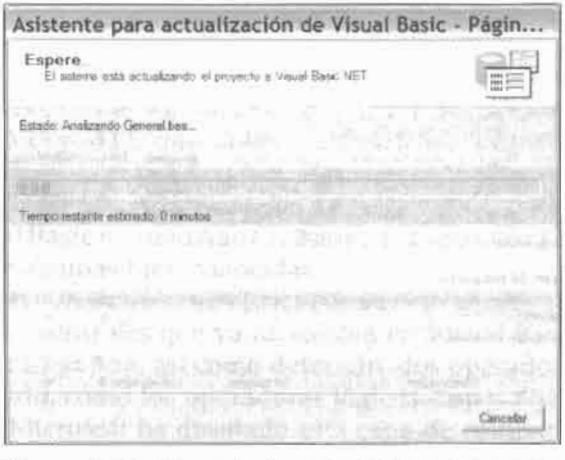


Figura 25.2. El asistente convirtiendo los módulos de nuestro proyecto

De encontrarse durante el proceso de conversión algún problema insalvable, como la inexistencia en el sistema de un control usado en el proyecto original, el asistente nos lo comunicará y detendrá el proceso. En ese caso concreto tendríamos que localizar el control en cuestión, registrarlo en el sistema y volver a poner en marcha el asistente de conversión.

Nota

Si nuestro proyecto es de una versión anterior a Visual Basic 6.0, por ejemplo de la versión 5.0, para efectuar adecuadamente la conversión tendre*mos que abrirlos primero con Visual Basic 6.0, guardarlos y finalmente* abrirlos con Visual Basic .NET.

Informe de actualización

Además de los formularios y módulos con código necesarios, según los elementos del proyecto original, al nuevo proyecto Visual Basic .NET también se añadirá una página HTML con un informe describiendo las incidencias encontradas durante el proceso de actualización. En él se enumeran los módulos que se han tratado, indicando su tipo, los problemas encontrados en él, etc. En nuestro caso, tras convertir el proyecto indicando antes, encontraremos un informe como el de la figura 25.3, sin ningún problema en la conversión a Visual Basic .NET. Es lógico dada la simplicidad del proyecto, pero no siempre será así.

La conversión no será, generalmente, tan directa y simple. En muchos casos tendremos que efectuar adaptaciones de forma manual, especialmente si el proyecto es de cierta complejidad. El informe no sólo nos notificará cuáles son los problemas sino que también nos guiará en el proceso de adaptación facilitando soluciones a problemas concretos.

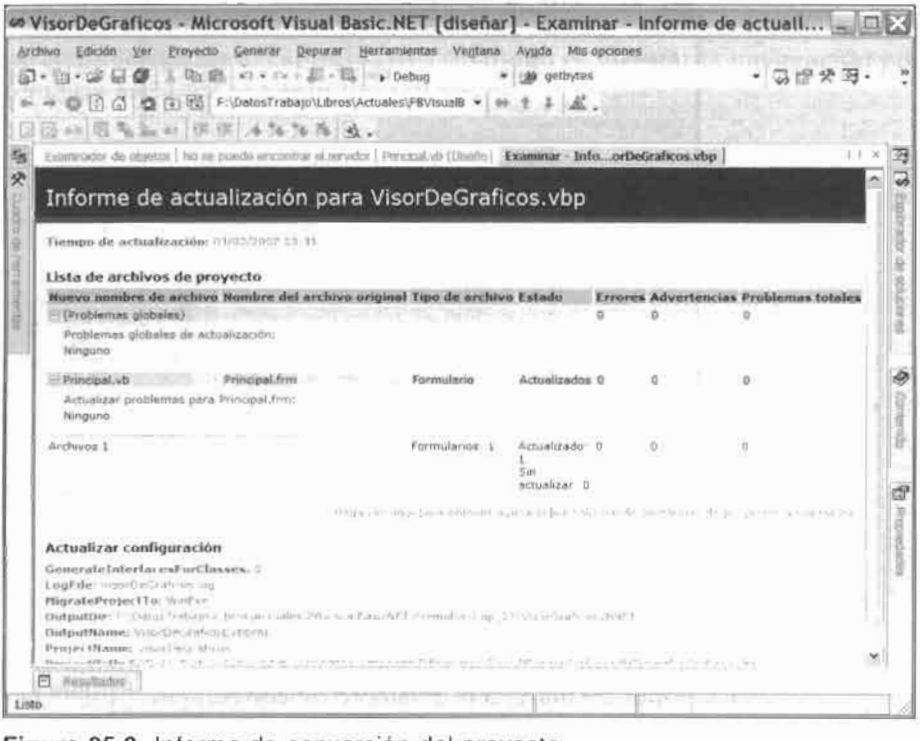


Figura 25.3. Informe de conversión del proyecto

La capa de compatibilidad de Visual Basic

Si tras ejecutar el asistente de conversión sobre un proyecto de Visual Basic 6 echa un vistazo al código obtenido en Visual Basic .NET quizá le sorprenda

ver algunas peculiaridades, como una sección de código adicional, en el caso de los formularios, que se encarga de crear la instancia por defecto que representa al formulario, o la aparición reiterada del ámbito `Microsoft.VisualBasic.Compatibility.VB6`.

Figura 25.4. Aspecto del programa escrito con Visual Basic 6.0 ejecutándose tras la conversión a Visual Basic .NET

Visual Basic .NET cuenta con dos ensamblados, alojados en los módulos `Microsoft.VisualBasic.dll` y `Microsoft.VisualBasic.Compatibility.dll`, que alojan gran parte de la funcionalidad de Visual Basic inexistente en la nueva versión. El primero de ellos, que se corresponde con el ámbito `Microsoft.VisualBasic` aloja multitud de funciones de uso habitual en Visual Basic 6, como `Asc()`, `Beep()`, `InputBox()`, `Mid()` o `Kill()`, por mencionar algunas bien conocidas.

En `Microsoft.VisualBasic.Compatibility.VB6` encontramos, además, controles que ya no existen en Visual Basic .NET, como `FileListBox` o `DirListBox`, así como determinados operadores y funciones que han desaparecido, como los operadores lógicos `Imp` y `Eqv`.

Microsoft ha diseñado esta capa de compatibilidad no para que utilicemos esas funciones, controles y operadores, algo que deberíamos evitar en los nuevos proyectos, sino con el fin primordial de hacer posible el funcionamiento del asistente de conversión de proyectos de Visual Basic 6 a Visual Basic .NET. Sin la ayuda de los dos ámbitos mencionados el trabajo de dicho asistente sería muchísimo más complejo y laborioso, al tiempo que el porcentaje de éxitos de conversión disminuiría. El código generado por el asistente de conversión, a pesar de que no se haya producido ninguna incidencia, generalmente puede ser mejorado de manera considerable evitando las referencias a la capa de compatibilidad, recurriendo en su lugar a los servicios de la plataforma .NET. Éste, sin embargo, es un trabajo que tendremos que efectuar manualmente.

Controles y llamadas al API

En los proyectos desarrollados con Visual Basic 6.0 es habitual usar no sólo los controles propios del producto, recurriéndose también a controles ActiveX

de terceros. Para poder convertir un proyecto de este tipo tendremos que instalar dichos controles en el ordenador donde actualmente tengamos Visual Basic .NET, registrándolos adecuadamente en el sistema. Si el asistente encuentra una referencia a un control que no está registrado la conversión no será posible.

Una alternativa es importar el control ActiveX desde Visual Basic .NET a posteriori, siguiendo los pasos que se describieron en el capítulo previo. Sería necesario, por tanto, efectuar la inserción manual de los controles en los contenedores y, posiblemente, retocar el código original para adaptarlo a la nueva configuración.

También el uso de funciones del API es una técnica habitual en Visual Basic 6.0, pero hay que tener en cuenta las diferencias entre los tipos de datos de esa versión y los del nuevo Visual Basic .NET. En la versión 6.0 si se espera un valor entero de 32 bits como respuesta de una función se utilizaba el tipo `Long`, mientras que en Visual Studio .NET el tipo correcto es `Integer` ya que `Long` es un entero de 64 bits. Esas diferencias puntuales son potenciales fuentes de problemas.

Habiendo leído y puesto en práctica los ejemplos del capítulo previo, no deberíamos tener problema alguno para utilizar un control ActiveX o convertir la llamada a una función del API de Windows en caso de que el asistente notificara algún tipo de problema.

Cambios obligados

A pesar de la existencia de la capa de compatibilidad citada antes, y de la equivalencia entre muchos de los controles de Visual Basic .NET respecto a los existentes en Visual Basic 6.0, existen ciertos elementos ante los cuales ni el asistente de actualización ni nosotros, de manera manual, podremos encontrar una solución efectiva y rápida. Algunos de esos elementos son:

- Acceso a datos con DAO o RDO – El método de acceso a datos en Visual Basic .NET, ya lo sabemos, es ADO.NET. Si en nuestro proyecto Visual Basic 6.0 usábamos ADO hay un camino para la actualización, pero si utilizábamos DAO y/o RDO no hay conversión posible, siendo trabajo nuestro la adecuación del proyecto.

- Uso de DDE – Las funciones del conjunto conocido como *Intercambio dinámico de datos* no están disponibles en Visual Basic .NET ni tienen un equivalente, por lo que en caso de haberlas usado en el proyecto original será preciso buscar otra alternativa para la comunicación entre aplicaciones.

- Arrastrar y soltar – Otra técnica habitual en Visual Basic 6.0 que no es convertida automáticamente a Visual Basic .NET mediante el asistente de actualización, quedando en nuestras manos.

- Clases web – En Visual Basic 6.0 las aplicaciones Web se basaban en el uso de unas clases que no tienen un equivalente directo en Visual Basic .NET.

Sería necesario generar una aplicación basada en formularios Web de ASP.NET y convertir manualmente la funcionalidad de un proyecto a otro.

- Control contenedor OLE – Este control, de uso muy habitual en Visual Basic 6.0 para facilitar la visualización y edición de documentos mediante OLE, no puede utilizarse en Visual Basic .NET, por lo que no hay conversión directa posible.
- Métodos gráficos – El asistente de actualización no es capaz de convertir automáticamente los métodos gráficos habituales de Visual Basic 6.0, como `Line`, `Circle` y `Point`, en sus equivalentes de GDI+. Otro trabajo que deberíamos realizar manualmente.

La anterior no es una lista exhaustiva de los problemas que podemos encontrar en la transición de Visual Basic 6.0 a Visual Basic .NET, sino simplemente una enumeración de los casos más habituales. En caso de que la conversión de sus proyectos actuales, por dimensiones e importancia, sea un factor primordial puede recurrir a documentación específica sobre el tema. La propia Microsoft edita un libro, titulado *Upgrading Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET*, en el que recorre a lo largo de casi 600 páginas toda la problemática de conversión de proyectos de una versión a otra del producto.

Puntos clave

- Visual Basic .NET dispone de un asistente que se activa automáticamente al abrir un proyecto de Visual Basic 6.0, facilitando su actualización automática y generando un informe de incidencias.
- La plataforma .NET dispone de unos ensamblados que contienen parte de la funcionalidad de Visual Basic 6.0 que ha desaparecido de Visual Basic .NET, haciendo posible el funcionamiento del asistente.
- El uso de controles ActiveX y las llamadas a funciones del API, así como el uso de ADO, RDO y ciertas técnicas y controles, no tienen una actualización automática y requiere siempre una intervención manual por nuestra parte.

Resumen

Como ha podido ver en este capítulo, a pesar de la existencia en Visual Basic .NET de un asistente de actualización lo cierto es que, en la práctica y con proyectos reales, la conversión de aplicaciones desde versiones previas de Visual Basic plantea diversos problemas que afectan desde al mecanismo de acceso a datos utilizado antaño hasta la imposibilidad de utilizar ciertos controles en la nueva versión.

La adaptación, por lo tanto, puede requerir un esfuerzo considerable, pero hay que evaluar también los beneficios que se obtienen al finalizar el proceso. Una vez que tengamos nuestra aplicación funcionando con Visual Basic .NET, posiblemente con ciertas dependencias de controles ActiveX, el API de Windows y la capa de compatibilidad mencionada, tenemos a nuestro alcance todos los servicios de la plataforma .NET, así como la tranquilidad de poder ir adaptando nuestro trabajo poco a poco, a medida que vayamos abandonando los servicios clásicos de Windows.